Framework Programmable Platform for the Advanced Software Development Workstation

Framework Processor Design Document

Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Dr. Richard J. Mayer, Thomas M. Blinn, Dr. Paula S.D. Mayer, Keith A. Ackley, Wes Crump and Les Sanders of Knowledge Based Systems, Inc. Dr. Charles McKay served as RICIS research coordinator.

Funding has been provided by Information Technology Division, Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Ernest M. Fridge, of the Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the authors and should not be interpreted as representative of the official policies, either express or implied, of NASA or the United States Government.

Framework Programmable Platform for the Advanced Software Development Workstation (FPP/ASDW)

Framework Processor Design Document

Produced For:

Software Technology Branch
NASA Johnson Space Center
Houston, TX 77058

Authors:

Dr. Richard J. Mayer
Thomas M. Blinn
Dr. Paula S.D. Mayer
Keith A. Ackley
Wes Crump
Les Sanders

Knowledge Based Systems, Inc.
2746 Longmire Drive
College Station, TX 77845-5424
(409) 696-7979

September 20, 1991

Table of Contents

List of Figures

List of Tables

1 Introduction

In this document, the design of the Framework Processor (FP) component of the Framework Programmable Software Development Platform (FPP) is described. The FPP is a project aimed at combining effective tool and data integration mechanisms with a model of the software development process in an intelligent integrated software development environment. Guided by the model, this Framework Processor will take advantage of an integrated operating environment to provide automated support for the management and control of the software development process so that costly mistakes during the development phase can be eliminated. This Platform is being developed under the Advanced Software Development Workstation (ASDW) Program sponsored by the Software Technology Branch at the NASA Johnson Space Center. The ASDW program is conducting research into development of advanced technologies for Computer Aided Software Engineering (CASE).

1.1 Motivations for the FPP

The FPP was conceived in response to the difficulties associated with producing software systems. With the advent of more powerful and more economical computer hardware resources, the complexity of software systems has increased dramatically. As computer systems become more complicated, ensuring that systems are consistently produced on time and within budget, while ensuring that the system built is reliable and maintainable, requires considerable management effort.

The large size of today's software systems makes it difficult for one person to fully understand the requirements, produce the design, and develop the system. Instead, the system development process must be executed by a team of managers and software engineers. Tasks within the development can occur concurrently except where certain tasks depend on information produced by others. These interrelationships make the management of the development process very complex. Regardless of how well a development project may be planned out, without some form of control over the actions of the development team costly mistakes and setbacks will occur during development. This is particularly true in multi-year projects that suffer from management and technical team leadership turnover.

Computer Aided Software Engineering (CASE) tools assist project managers in both monitoring the progress of the development activities and in capturing the experiences of the development team. However, existing CASE tools fail to cover the entire software development process and concentrate instead on a particular aspect of the development process (i.e., project management, requirements analysis, code development and debugging). The result has usually been to use a piecemeal collection of various CASE tools that addresses only portions of the software development process.

It is observed that most CASE tools are useful within a specified area of the system development process. A persistent problem, however, is in using these tools in an organized fashion so as to fully automate the system development process. Incompatible data formats along with the misuse of tools make interaction among these different tools very difficult. As a result, CASE environments that effectively automate the software engineering process have yet to be developed.

The recognition of these difficulties has spurred the development of the FPP. The focus of the FPP is the management, control, and integration of the software system development process. The major objectives in this definition of the FPP have been to provide:

1) a realistic integration strategy that supports function and data integration of a suite of tools (distributed and covering the entire life cycle);
2) integrated access to and update of life cycle artifact data;
3) control of life cycle activities and data evolution; and
4) a site-specific development process support environment enforcing the rules and preferred methods of the organization.

The FPP is also expected to provide these capabilities in a distributed, heterogeneous computing environment. Developing a platform that meets these objectives will result in (1) a reduction in the time required to produce software systems, (2) an increase in the quality of the resulting software systems, (3) a decrease in the maintenance effort for the resulting software systems, and (4) an increase in the consistency in the development process by which software systems are constructed.

1.2 Scope of this Document

Prior to the work presented in this document, work performed by KBSI related to the FPP focused on defining how the FPP should operate at a conceptual level and then reducing those concepts to functional requirements. As the conceptual design and requirements definition have been completed ([FPP 90a], [FPP 90b]), the FPP project is now well into the design phase of the project. During this design phase the focus is on the definition of how operations and capabilities identified in the Concept Document are to be provided by components of the FPP.

The focus of this document is the Framework Processor (FP) component of the FPP. This component is responsible for capturing and processing a description of an organization's software system development process and for using that description to provide automated support for the control and management of that development process. It is through this process description and the Framework Processor that the FPP gains its programmability. The framework (process) definition can be modified to reflect new knowledge about the development process or changes in the process and those changes will be incorporated into the development environment.

A description of the mechanisms by which the Framework Processor will represent, manage, and use the framework process description to control and manage the software development process will be detailed in this document. Also detailed in these descriptions will be the means by which the Framework Processor will interact with elements of the FPP software integration platform (i.e., the remaining elements of the FPP) as well as how the Framework Processor could interface with other integration environments.

1.3 The Design Approach

As the FPP is a large system whose design involves many issues concerning integration and the development process, an iterative design approach is being used. This iterative approach will allow the design team to examine particular aspects of the FPP while making certain assumptions about other components of the platform. As designs of components are completed and new components are examined and detailed, the previous designs will be re-examined to determine if the assumptions made during the design of that component still hold.

This document represents the design of the Framework Processor, the second component addressed as part of the design of the FPP. The initial design document [FPP 91] centered on the Integration Mechanism component of the FPP. In light of the design process described above, the design described in this document can be considered a "living" design. This means that revisions can, and probably will, be made to this design. While the changes that may occur will expand and clarify areas where the current design may be lacking, major revisions to the Framework Processor design are not expected.

1.4 Document Organization

Discussing the design of the Framework Processor requires discussion of several topics: (1) the concepts behind framework processing, (2) the structure of the framework, and (3) the components required to manipulate and use the framework definition.

The discussion begins in Section 2 with a presentation of the role frameworks can play in the software development process and strategies for using the framework to provide automated support for software development. An understanding of these topics is necessary to follow the design of the Framework Processor as the concepts discussed in this section lay the groundwork for the framework structure and framework processing capabilities discussed in later sections.

Section 3 is dedicated to the description of the syntax for and structure of the framework definition. This discussion reflects a major part of the design process as the structure of the framework representation had to ensure that the expressive capabilities of the framework were not limited. To limit the framework specification would make it difficult to capture the complexities of the software development process and thus limit the capabilities of the Framework Processor.

The design of the Framework Processor is presented in Section 4. A major part of the operation of the Framework Processor revolves around constraints and constraint propagation. A considerable part of this discussion describes how the structures that make up the Framework Definition (Section 3) are transformed into constraints and then used by the Framework Processor to manage the software development process. The other components of the Framework Processor are also discussed. A scenario of operation then shows how the various pieces of the Framework Processor fit together.

Finally, Section 5 relates the Framework Processor design to the other components of the FPP and discusses areas where future work on the Framework Processor could be directed. Section 5 is followed by two Appendices. Appendix A presents a list of acronyms used in this document, and Appendix B shows the grammar for the Definition Component and the Elaboration Specification of the Framework Definition.

2 Framework Processor Concept

Before a discussion of how the Framework Processor (FP) accomplishes its task, an overview of what the Framework Processor is intended to do must be provided. A major goal of the FPP project is to capture a description of an organization's software development process and to use that description to monitor and control the actual development of software systems. Within the context of the FPP project this system development process description is captured in a framework. The purpose of the Framework Processor is to process and interpret this framework and to use the information stored in the framework to control an organization's development projects. It is through the information captured in the framework and the functionality provided by the Framework Processor that the FPP will be "programmed." The framework will serve as the program or source code while the Framework Processor will act as a compiler and/or interpreter.

2.1 The Development Framework

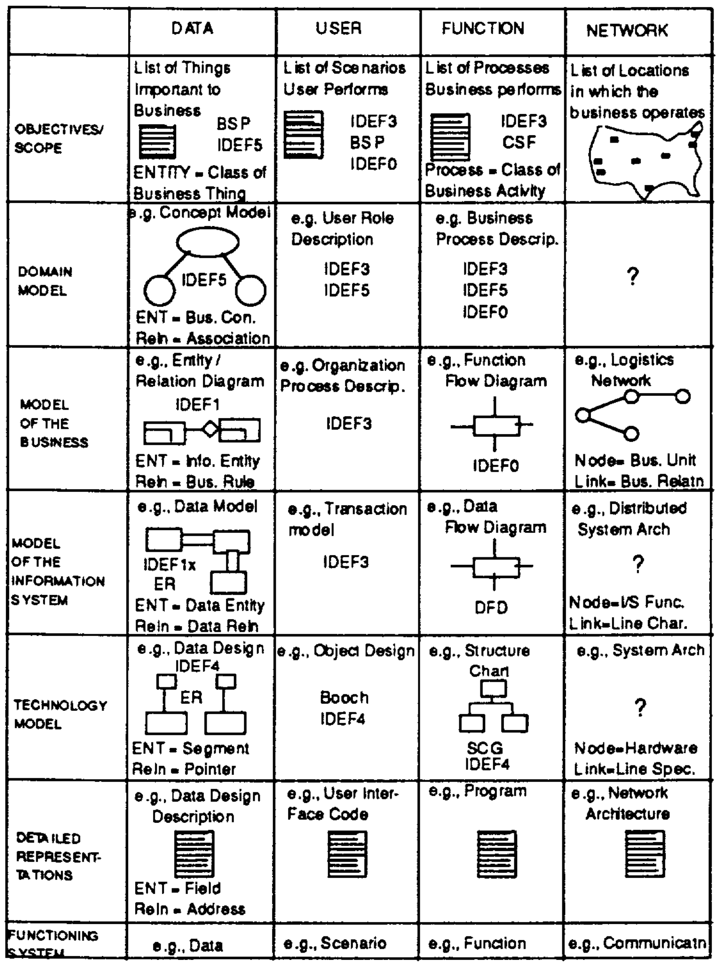
The overall framework to be processed by the Framework Processor is made up of two overlapping views. One view is the Situation Classification Framework (SCF) view and is concerned with the identification and definition of specific development situations.[1] The other view is the System Development Framework (SDF) and is concerned with life cycle analysis, design, implementation, maintenance, and decision-making activities. The following two subsections will discuss the nature and content of these two views. More detailed discussions of the content of the framework can be found in Section 3.

2.1.1 Situation Classification Framework

In the development of software systems, different representations of a system architecture are developed at different stages of the development process. The Situation Classification Framework attempts to characterize the development situations that require the different representations. This characterization includes identification of the roles, responsibilities, conditions, prior commitments, and information involved in a situation that results in a need for a particular class of representation. This necessary representation can then drive the selection of specific methods for capturing that representation.[2]

[1] In the FPP Concept of Operations Document [FPP 90a], the SCF was described as the Method Classification Framework (MCF). In our research since the production of that document, the scope of this framework view has broadened to address development situations rather than just the methods used in those situations.

[2] It was because of this selection of methods that this framework view was originally labelled as a Method Classification Framework.



	DATA	USER	FUNCTION	NETWORK
OBJECTIVES/ SCOPE	List of Things Important to Business BSP IDEF5 ENTITY = Class of Business Thing	List of Scenarios User Performs IDEF3 BSP IDEF0	List of Processes Business performs IDEF3 CSF Process = Class of Business Activity	List of Locations in which the business operates
DOMAIN MODEL	e.g. Concept Model IDEF5 ENT = Bus. Con. Reln = Association	e.g. User Role Description IDEF3 IDEF5	e.g. Business Process Descrip. IDEF3 IDEF5 IDEF0	?
MODEL OF THE BUSINESS	e.g., Entity / Relation Diagram IDEF1 ENT = Info. Entity Reln = Bus. Rule	e.g. Organization Process Descrip. IDEF3	e.g., Function Flow Diagram IDEF0	e.g., Logistics Network Node= Bus. Unit Link= Bus. Relatn
MODEL OF THE INFORMATION SYSTEM	e.g., Data Model IDEF1x ER ENT = Data Entity Reln = Data Reln	e.g., Transaction model IDEF3	e.g., Data Flow Diagram DFD	e.g., Distributed System Arch ? Node=I/S Func. Link=Line Char.
TECHNOLOGY MODEL	e.g., Data Design IDEF4 ER ENT = Segment Reln = Pointer	e.g., Object Design Booch IDEF4	e.g., Structure Chart SCG IDEF4	e.g., System Arch ? Node=Hardware Link=Line Spec.
DETAILED REPRESENT-TATIONS	e.g., Data Design Description ENT = Field Reln = Address	e.g., User Inter-Face Code	e.g., Program	e.g., Network Architecture
FUNCTIONING SYSTEM	e.g., Data	e.g., Scenario	e.g., Function	e.g., Communicatn

Figure 1. The Zachman Framework

An example of a Situation Classification Framework derived from John Zachman's original framework [Zachman 86] is shown in Figure 1. The framework is represented as a matrix in which the six rows represent different *perspectives* (or views) and the four columns represent *focuses* of descriptions of an information system architecture. The perspective organizes the descriptions of the system architecture with respect to multiple viewpoints (e.g., the executive, the manager, the programmer, etc.). The focus organizes the descriptions with respect to the level at which the system will operate. Thus, each cell in the matrix represents a situation with a particular focus from the perspective of a user's viewpoint.

For example, the framework illustrated in Figure 2 involves members of an enterprise working to develop an information system that will provide the data required to evaluate the corporation's critical success factors. To this end, the corporate manager is trying to identify information needed to determine if his/her corporation is meeting the critical success factors of the enterprise. The business manager is trying to identify information needed to determine if his/her business is meeting the goals passed down from corporate. So, both people are taking the same focus (i.e., the data or information focus), but from two different perspectives. For this example, the general situation type for the data column is to analyze the information requirements necessary to evaluate performance measures at the various levels of the organization.

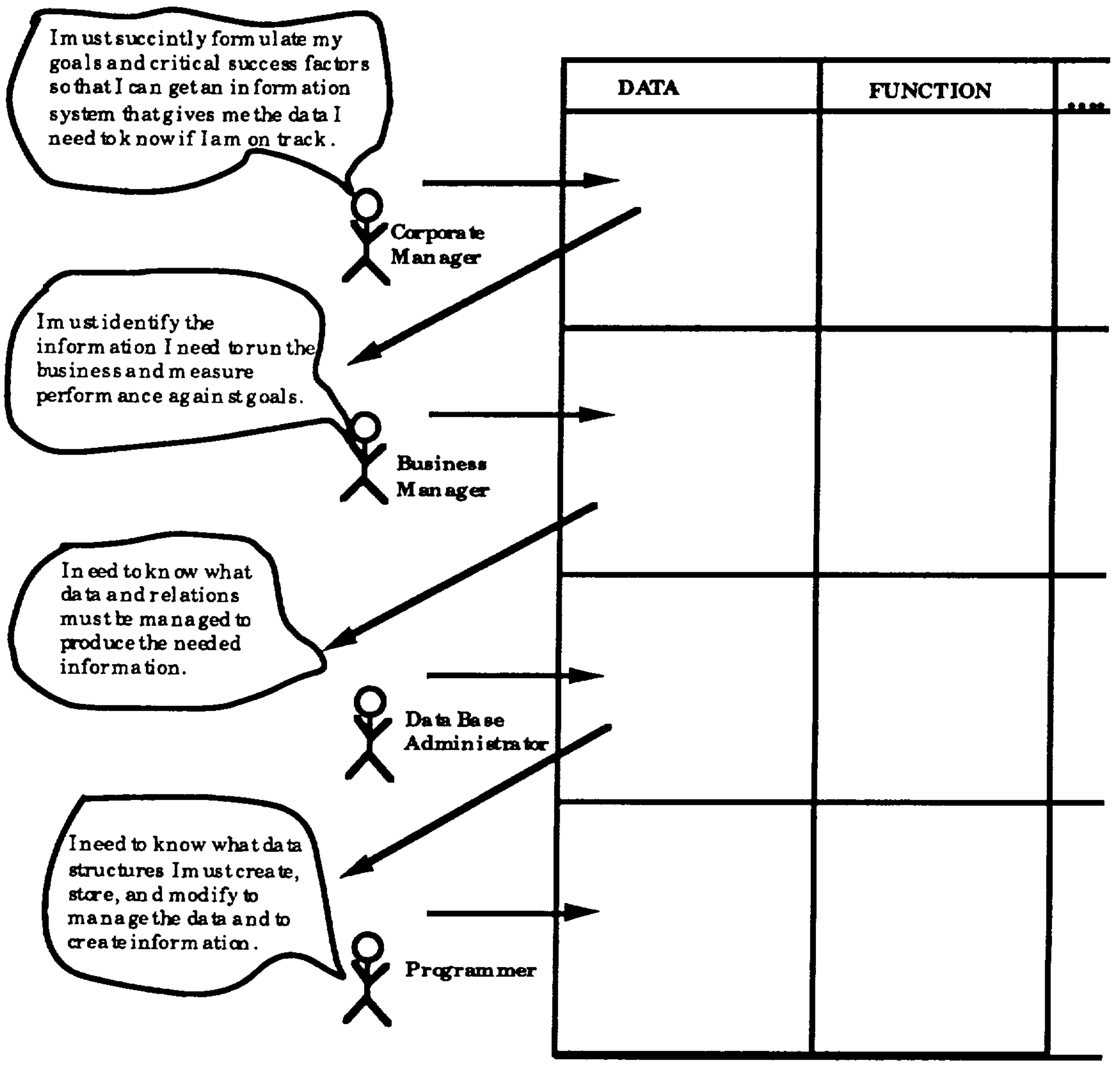

I must succintly formulate my goals and critical success factors so that I can get an information system that gives me the data I need to know if I am on track.
Corporate Manager
I must identify the information I need to run the business and measure performance against goals.
Business Manager
I need to know what data and relations must be managed to produce the needed information.
Data Base Administrator
I need to know what data structures I must create, store, and modify to manage the data and to create information.
Programmer
DATA
FUNCTION
...

Figure 2. Development Process Situation Types

Each cell in a framework represents a characterization of reoccurring development situations in an organization. As such, the Situation Classification Framework provides a means to carry the experience base from one project to another within an organization. In addition, the framework can provide a degree of control over the system development and provide consistency between projects requiring multiple project coordination, management consistency, and personnel utilization.

2.1.2 System Development Framework

While the Situation Classification Framework view attempts to categorize the development situations that arise during system development, the Situation Classification Framework provides no means for capturing temporal relationships between the various situations. In addition, there are no means for capturing the details of the processes and activities that occur within the situation types. With the System Development Framework on the other hand, the intent is to capture these procedural aspects of an organization's system development process.

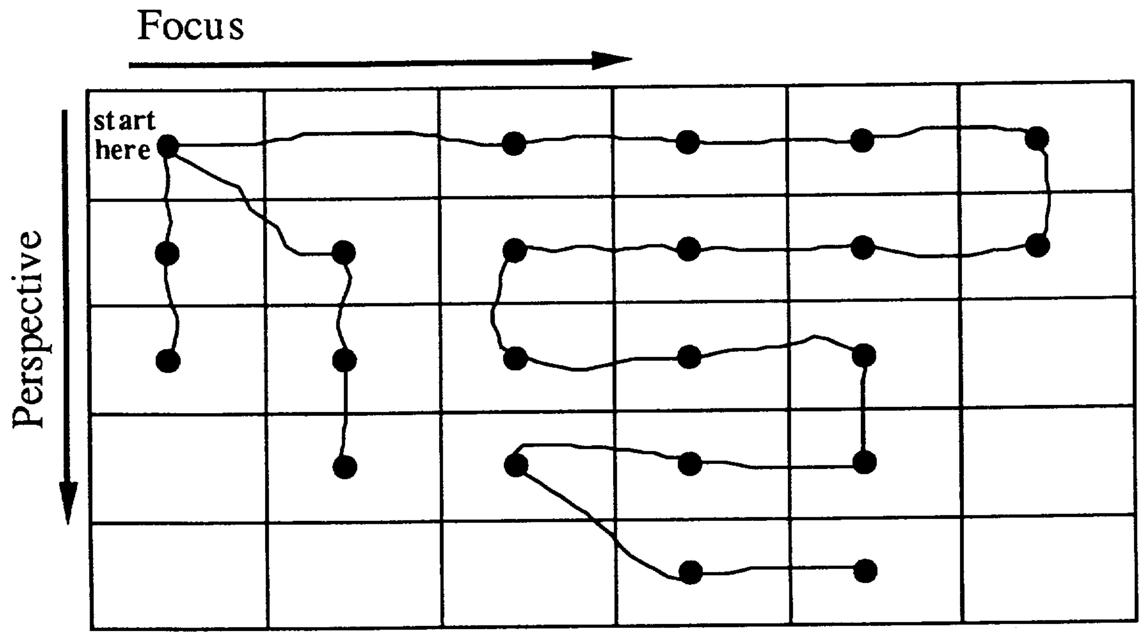


Figure 3. Precedence Relationships Between Development Situations

The System Development Framework view of the FPP framework can be seen to have two process components. The first is the overall system development process description that attempts to define the sequence of situations that are encountered during the development process. Figure 3 illustrates precedence/temporal relationships defined between the various cells of the Situation Classification Framework. Within the Situation Classification Framework, there is no sequence or ordering to the cells of the framework. However, in the specialization of the framework for an organization, the definition of the temporal relations between development situations will be an important activity. Notice in the figure that some cells are "visited" while other cells are ignored. It is likely that the set of visited cells and the sequencing of those visited cells will be different for different classes of system development activities or for different organizations.

While the first process component addresses the overall development process, the second process component of the System Development Framework describes the specific activities necessary to address the development situations represented by the individual cells of the Situation Classification Framework. These system development process definitions include not only the life cycle phases, tasks, milestones, and documentation artifacts, but also:

1) descriptions of the procedures for analysis, decision making, and configuration control;
2) calls for the application of specific methods;
3) definition of common information/data across the different methods;
4) descriptions of how method results will be applied; and
5) role definitions.

Together, these data provide a complete description of the process by which an organization addresses the development situations represented by the Situation Classification Framework.

IDEF3, the method chosen to represent the development processes, has the necessary features for capturing both the above process description components. With IDEF3, each of the situation types characterized by the cells in the Situation Classification Framework can be represented as UOBs (Units of Behavior) at the overall system development process level. The sequencing and the relative timing of these situations can be represented in the IDEF3 diagrams using precedence links. IDEF3's ability to decompose UOBs will facilitate the description of the activities necessary to address each of the high-level situations.

2.1.3 The Combined Framework

The preceding discussion has presented the two framework views as being separate structures. In actuality, the two frameworks are closely linked. By moving down a level of abstraction from the Situation Classification Framework, it is apparent that each cell of the Situation Classification Framework points to more information as shown in Figure 4. Part of this information is the process description that captures details of the activities involved in addressing the situation. Therefore, the System Development Framework is partitioned and distributed across the Situation Classification Framework. The FPP will take this approach towards the framework definition. The Situation Classification Framework will serve as an organizing structure for the information necessary to capture an organization's development process.

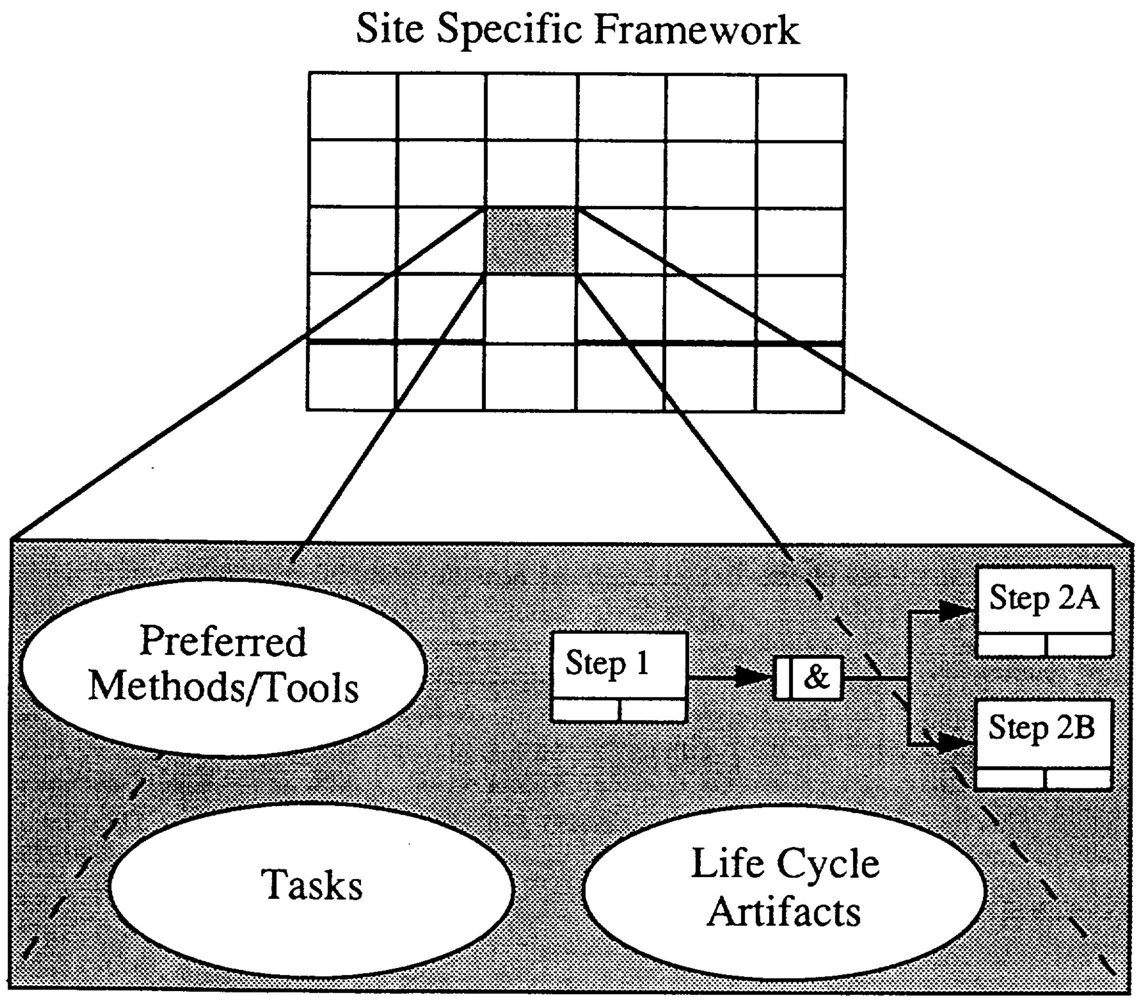


Figure 4. The FPP Framework

Taken together, this overall framework provides structure for the description of the software development process and:

1) provides a "big picture" of the system development process;
2) provides a "quick road map" for the participants in the system development process;
3) identifies standard methods and tools;
4) specifies applicable tools and methods at a site;
5) assists in the planning and scheduling of the system development process;
6) orchestrates the use of integrated tools and methods; and
7) summarizes the standard development process at a site.

Once a framework has been defined, the opportunities for use of this knowledge base are almost limitless. Some of the capabilities that will be possible through the use of this approach are:

1) Context Defined Tasking,
2) Life Cycle Data Management and Control,
3) Automated Project Status Reporting,
4) Documentation Generation, and
5) Automatic Problem Notification.

These types of capabilities are possible because the framework completely defines the activities that will occur during the development process, the relationships between those activities, the objects (e.g., documents, code, and modules) that will be manipulated during a particular activity, and the roles of people that will be involved in the activity.

2.1.4 The Framework Cell Definition Process

At first glance, it appears simple to define a framework for an organization. However, once one starts puzzling over the individual cells, three issues are quickly recognized:

1) Only experienced system developers understand the recurring situations and complex interactions between the roles and objects of interest that must be represented in the framework.
2) Accurately describing each situation in a manner that is clearly understood by others involved (both directly and indirectly) in the development process is non-trivial.
3) Reference frameworks are adequate starting points, but never quite fit the site specific situation upon close inspection.

One of the promising approaches for specializing site specific frameworks from reference frameworks is based on the observation that one way to characterize a situation type is by identifying the questions that should be answered by a properly executed instance of that situation type. Rather than trying to initially identify the situations, roles, objects, and relations of interest, it is often easier to collect the questions that we would like to answer from the artifacts produced by the (yet unidentified) situation types. This approach is consistent with Zachman's initial intuition that the cells represent descriptions of the envisioned (or actual) system that reduce risk by explicitly documenting the various decisions and views of the system. That is, if I want to know why a particular piece of data exists in the system, I should be able to find a business performance or operational information requirement that that data supports directly (or a system design decision that requires that data for operation of the system that ultimately supports that business requirement).

Upon examination of a typical framework, we can classify general question types, and we can identify general question templates within each question type. One such question type is referred to as an "introspective" question.

Introspective questions are questions associated with a cell that are directed at the personnel in the organization who have the perspective indicated in the row label of the cell. An example of such an introspective question is "What are the goals of the enterprise that are affected by the system?" Such a question would be associated with the cell in row one and column one of the Zachman framework shown in Figure 1. It happens that such a question is also a good example of an instance of a reoccurring question schema. One of the general templates for introspective questions is "<what, who, where, when> <be verb form> the <column focus> of the <row perspective> of the <system name>?"

Once the question templates have been specialized for a site, the next step in the cell formulation is the definition of the situations within which such questions could be answered. The roles of such a situation definition would include those types of personnel either responsible for getting the answer or in possession of the information that the question is asking for. The objects of interest often represent the elements of the answer to the question, as do the object relations. Once the situation descriptions have been formulated, the next step is to specify the rules that govern the question answering process in such a situation. These rules can reference the results of other cells as well as the objects and roles of a particular cell. The process definition itself would take the form of a set of IDEF3 process descriptions. It is also at this stage that the selection of methods (and tools) to support this process application (and rule enforcement) would be accomplished. Finally, the rules for how the answers to the questions are to be used and managed must be defined. This process can be accelerated by having access to generic or reference frameworks.

In this project, the process description captured by the framework will be used to monitor, manage, and control development projects. To ensure that the Framework Processor operates correctly requires that the framework accurately reflects the actual process followed or desired by the organization. The operation of the Framework Processor will only be as valuable as the information maintained in the framework. Because of the importance the FPP will place on the defined framework and of the inherent complexity of frameworks, the task of defining a framework for a specific organization should not be taken lightly.

2.2 Framework Processor Architecture

The previous section has provided an overview of the basic content of a framework from the perspective of the FPP project. This section will now detail how the Framework Processor will use the framework to provide automated support for the management and control of the software development process. The description will begin with a conceptual discussion of the relationship between the Framework Processor and the Integration Platform, the software development environment used at a particular organization. This discussion is then followed by a description of the functional architecture of the Framework Processor.

2.2.1 Framework Processor Conceptual Architecture

The Framework Processor is intended to serve as a controlling mechanism over the software system development process. This requires the Framework Processor to interact with an organization's development environment (platform). During the evolution of the Framework Programmable Platform (FPP) project, the relationship between the Framework Processor and the actual integration platform / environment has been studied from many different views. An initial view is shown in Figure 5. In this view, the Framework Processor plays a passive role by simply processing the framework specification and "loading" a set of knowledge bases. These knowledge bases are then accessed by the Integration Platform to monitor and control the development process.

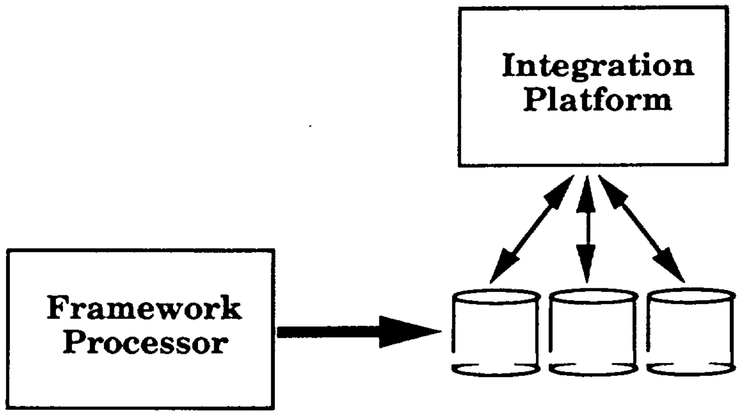


Figure 5. Initial Framework Processor / Platform Relationship

A more detailed architecture for this relationship is provided in Figure 6. The Framework Processor would take as input the framework definition and translate the it to a neutral representation. This neutral representation could then be accessed by an integration platform, through an appropriate interface.

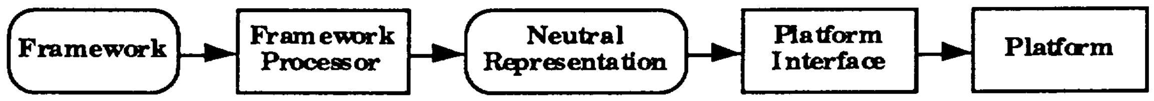


Figure 6. A More Detailed View of the Initial Relationship

This view reflects a very close relationship between the information represented in the development framework and the integration platform and places the burden for understanding and the contents of the framework and for using the framework information to control the development process on the integration platform. As a result, the Integration Platform was tied very tightly to the structure and content of the framework, and the framework was tightly coupled to the functionality of the Integration Platform.

However, the evolution of numerous integration efforts and products have prompted the development of a more general view for the Framework Processor. This view requires the Framework Processor to take a more active role in the controlling process. A conceptualization of this new view is reflected in Figure 7. Upon initial examination, this view does not appear to be very different from the previous views. However, differences do exist. The main difference is that the interfaces now lie between the Framework Processor and the Integration Platform as opposed to the previous view where the interfaces were defined between the Integration Platform and the knowledge base containing the neutral representation. As such, these new interfaces require a degree of cooperation between the Framework Processor and the Integration Platform and will provide the means by which an Integration Platform can send messages to the Framework Processor about operations performed as part of the development process. Therefore, the Integration Platform will notify the Framework Processor of the completion of certain events. The Framework Processor will then determine whether that event has any effect on currently active development projects.

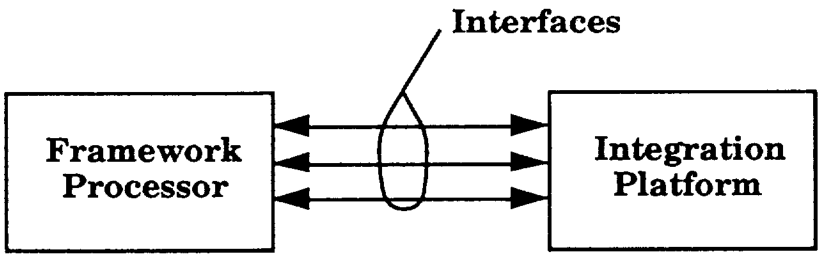


Figure 7. The Generalized Relationship

Although this view allows the Framework Processor to be applied to many different integration platforms, the capabilities of the Framework Processor might be limited by the number of interfaces the particular integration platform supports. To its advantage, though, this view will not require the integration platform to manipulate the information represented by the framework in order to manage and control the development process. For this reason, the design of the Framework Processor is based upon this second conceptual view.

2.2.2 Framework Processor Functional Architecture

The previous section has addressed, at a high level, how the Framework Processor will operate and interact with integration platforms. Essentially, the Framework Processor lies between a framework definition and an organization's development environment (integration platform). This section will address, still at a relatively high level, the functionality required to operate between the framework and the environment. This functional description will be provided by identifying and discussing the components that make up the Framework Processor and the roles that those components play in processing and managing a framework definition. More detailed discussion on the operation and specification of each of these components will follow later in this document.

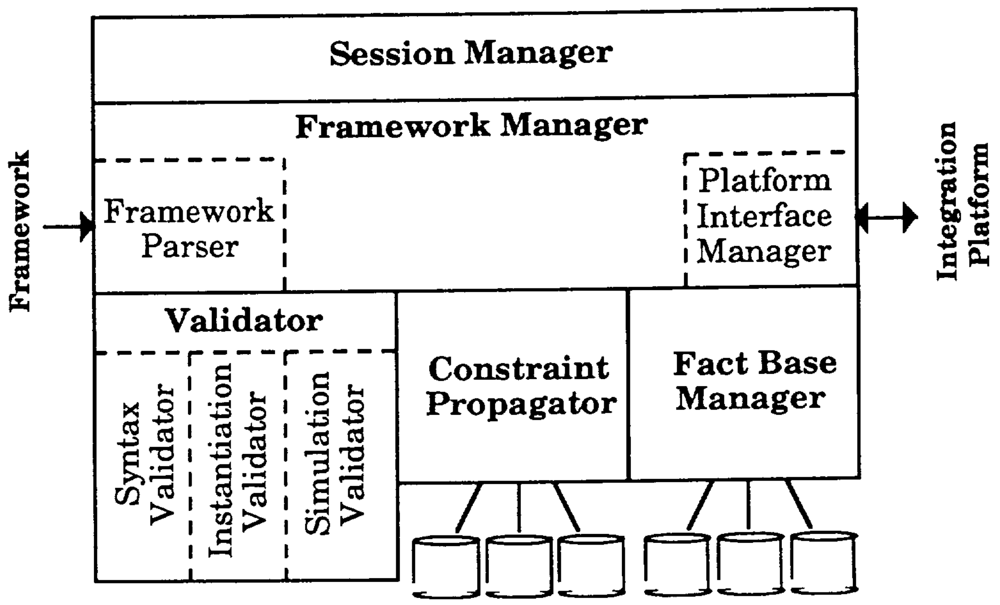


Figure 8. Framework Processor Architecture

Figure 8 presents the functional architecture for the Framework Processor. The basic operational philosophy of the Framework Processor is to take a framework as input, perform several validation checks on the framework, translate the framework into a set of constraints and facts, and then use the facts and constraints to control the development process. During this process, the set of facts and constraints are continuously updated as a result of actions by users and messages from the integration platform (i.e., the notification of the occurrence of certain events). This dynamic situation is continuously monitored to detect inconsistences between the process specified in the framework and the actual events occurring during the system development.

The overall operation of the Framework Processor is controlled by the Framework Manager. Prior to framework installation, the Framework Manager coordinates the framework validation process by parsing the framework and extracting the appropriate information for the Validator component. Three levels of validation are performed by the Validator component:

1) syntax - to ensure the IDEF3 descriptions included in the framework definition adhere to the syntax rules of the method;
2) instantiation - to ensure that a realization of the process represented by the framework does not contain any inconsistencies (e.g., at one point in the framework

specifying a constraint that is contradicted at another point in the framework); and

3) simulation - to detect inconsistencies that cannot be detected at instantiation by simulating potential process scenarios.

Each of these validation steps are complex enough to require an individual sub-component within the Validator. In the event that inconsistencies are detected, the Framework Manager would assist the framework administrator in correcting the inconsistencies.

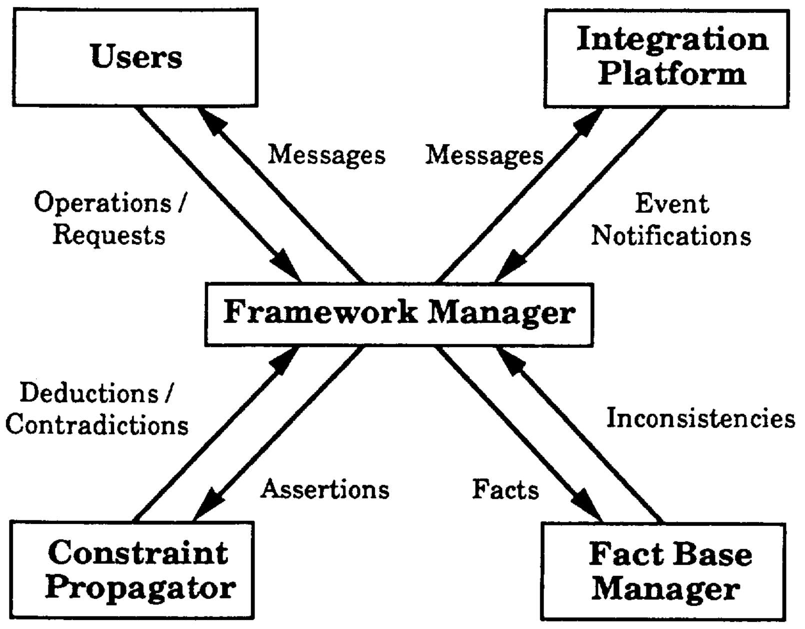


Figure 9. Framework Manager Operation

After validation has been completed, the framework is installed and project instantiations can be performed. Once a project has been instantiated, the Framework Manager begins to monitor and control that project development. Figure 9 shows how this monitoring is performed. At project instantiation, an initial set of assertions are passed to the constraint propagator and a set of facts are passed to the Fact Base Manager. From that point, information about operations and events performed and requests for authorization by project members are continuously passed to the Framework Manager from the users (through the Session Manager) and the Integration Platform. This information is then passed to the appropriate knowledge base (i.e., constraint base or fact base). If contradictions or inconsistencies are detected, it is up to the Framework Manager to take appropriate action to resolve the conflict. If no problem is detected, the operations are performed, and the project state is updated.

Notice that the knowledge bases have been split between facts and constraints, and accordingly two different components have been devised to manage those knowledge bases. The reason for this is that the information represented in the framework is so diversified that using a single reasoning scheme became impractical. Instead, the fact base and the Fact Base Manager capture and manage information about access privileges, users, user roles, etc., while the constraint base and the Constraint Propagator contain and maintain the state of the project development process. As such, the Fact Base Manager processes the static framework information, while the Constraint Propagator processes the dynamic process-oriented framework information.

Interaction between the users and the Framework Processor can occur either by direct interaction or indirect interaction. Direct interaction will be managed by the Session Manager. This component will, for the common user, provide capabilities for logging into a specific project, checking the status of the project, inquiring about open tasks, and browsing the system development process. For special users like framework administrators and project managers, the Session Manager will provide the mechanisms for installing frameworks and instantiating projects based on a particular framework. Indirect interaction with the Framework Processor can be achieved through the Integration Platform. The architecture for the Framework Processor supports interfaces between itself and the Integration Platform. If the platform supports these interfaces, the platform can send messages to the Framework Platform to indicate the performance of certain operations by the user. In a similar manner, the Framework Processor can send messages detailing the consequences of those events back through the Integration Platform to the user. This ability to support indirect interaction depends on the degree to which an organization's development environment is integrated with the Framework Processor.

3 Framework Definition

As was mentioned previously, the framework definition is a complex process. Just as important, however, is the representation of the framework contents. The information describing the development process must be structured in a format that will allow efficient processing by the Framework Processor so that the Framework Process can enforce the policies and procedures captured in the framework. The purpose of this section is to define the organization and structure of the information within the framework. Essentially, this definition provides the syntax for the framework definition.

In the previous section, the representation of the knowledge about the development process was partitioned into two frameworks: the Situation Classification Framework and the System Development Framework. In the specification of the framework structure to follow, these two framework views are integrated into one logical framework structure. Figure 10 shows how the two concepts of classification and process can be combined. In this form, the situation cells become UOBs and the matrix transforms from a static classification scheme into a dynamic process description of the software development.

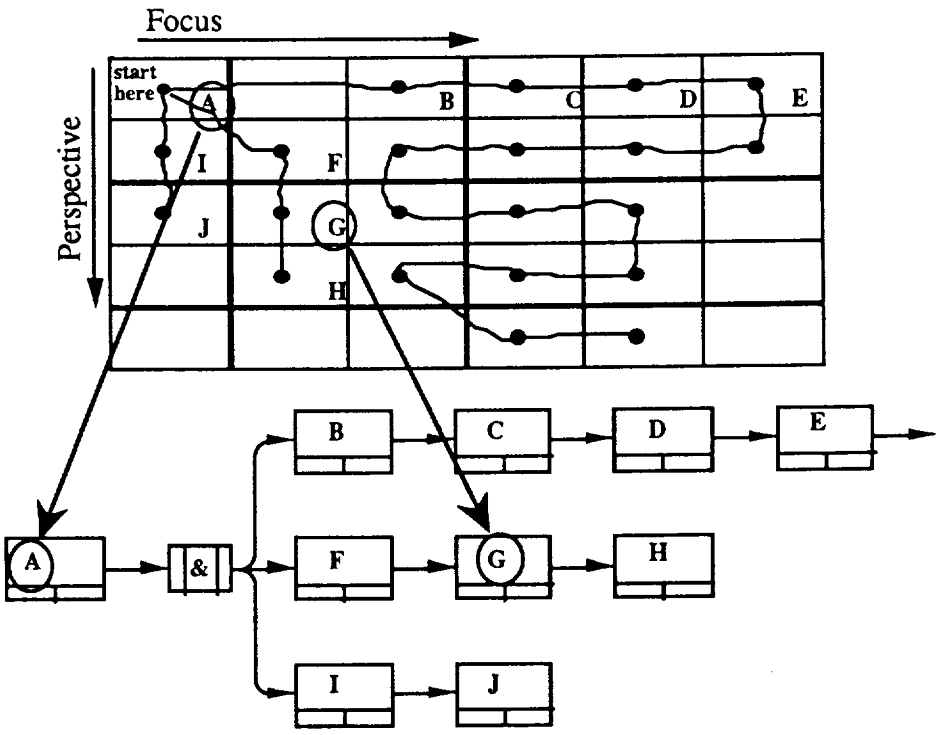


Figure 10. The Framework Translated to IDEF3

The framework is the medium for providing system development knowledge to the Framework Processor. Thus, the framework is much

PRECEDING PAGE BLANK NOT FILMED

more than just a matrix containing the names of methods and artifacts. Instead, it is the collection and organization of all of the situations that make up the software development process along with the processes, tools, methods, user roles (or types), and artifacts that are instrumental in the development of a software system. For the purpose of the Framework Processor, the definition of the framework structure breaks down into two major components:

1) The *Definition Component* defines the acceptable vocabulary for the framework representation, in essence the ontology of the development process;
2) the *Process Specification Component* defines the system development process and the facts and constraints pertaining to that process.

The following sections will further define the structure this framework medium must take and will specify the types of information that should be represented in the framework.

3.1 Definition Component

The Definition Component establishes the vocabulary for use within the Process Specification Component. The advantage of a common vocabulary lies in the ability to standardize the framework definition on a common set of terms, thus providing the means for the Framework Processor to validate the framework definition by preventing completely free-form constraints. The Definition Component supplies the ontology for the framework as it identifies those object types that will exist within the development process and whose instances can be referenced in the framework definition.

These objects fall within broad types of which there are currently five identified, including:

1) a User Role, which identifies a class of project members;
2) an Artifact, which identifies a product of the development process;
3) a Tool, which identifies a tool running in the development environment;
4) a Method, which identifies a method used by an organization; and
5) a Relation, which identifies a user specified relationship.

During the definition of the framework structure, it was determined that some objects exist outside the scope of the framework. Despite this fact, knowledge about the existence of these objects is required for the Framework Processor to function properly. For this reason, these object types are included in this discussion. The two object types presently identified are:

6) a User, which identifies an individual within the organization and
7) a Project, which identifies a current development effort.

The statements defining the objects that make up this vocabulary are captured in a set of definitional forms and are discussed in the following subsections.

3.1.1 User Roles

Much of the control of the software development process is dependent on the role played by the user. The users of the FPP will have different responsibilities and access privileges depending on their function in the development process. The Framework Processor must have some way of distinguishing these differences for it to manage the development. This is accomplished by defining user roles for the site. The syntax for defining the user roles is as follows:

```
(DEFUSERROLE        :NAME symbol)
```

where *symbol* represents a user role. This defines a user role and allows its use in relations, facts, and constraints within the framework. For example, the user role `programmer` is defined by the following:

```
(DEFUSERROLE        :NAME programmer)
```

When the framework is instantiated, `programmer` will be a valid user role and can be used to constrain access to artifacts and processes.

3.1.2 Artifacts

Any large software development effort will generate a large number of documents, reports, manuals, code modules, and other objects, otherwise known as artifacts.[3] These artifacts can progress through different states of evolution during their life cycle. The Framework Processor must know the valid states in which an artifact may exist throughout the development process. In addition, the Framework Processor must know what tools can be used to evolve the artifact.

This information is provided in a DEFARTIFACT statement. The syntax for this statement is shown below:

[3] Although artifacts can exist in paper form, throughout this document the term artifact refers only to the electronic media representation, as this is the only form to which the Framework Processor can effectively control access.

```
(DEFARTIFACT :NAME       symbol
             :TOOLS      (tool0, tool1, ..., tooln)
             :METHODS    (method0, method1, ..., methodn)
             :STATES     (state0, state1, ..., staten))
```

where *symbol* is the unique name for an artifact. *Tool*$_0$, *tool*$_1$, ..., *tool*$_n$ are names of tools that can be used to evolve the artifact. *Method*$_0$, *method*$_1$, ..., *method*$_n$ are the names of methods that can be used to capture the information contained in the artifact. *State*$_0$, *state*$_1$, ..., *state*$_n$ are the names of the valid states that the artifact can have during its evolution. An example of how to define the SoftwareTestReport using the DEFARTIFACT statement is shown below:

```
(DEFARTIFACT :NAME       SoftwareTestReport
             :TOOLS      (MSWord, WP50, Emacs)
             :METHODS    ()
             :STATES     (Uncreated, Created, InProgress,
                              InReview, Rework, Completed))
```

It should be noted that an artifact will have other attributes that are required by the Integration Platform for both storage and tracking that can be accessed by the Framework Processor as needed. However, this information is not placed into the Framework Processor's knowledge bases.

3.1.3 Tools

Tools are used by the software development team to create, view, and modify artifacts required by the framework definition. A tool may or may not have a method associated with it. For example, a word processing tool does not have a method, whereas Automated IDEF0 (AI0) supports the IDEF0 function modeling method. A tool may be capable of supporting more than one method. A tool supports a set of file formats for saving the information it captures. By combining these facts, resulted in the development of the following form for defining tools:

```
(DEFTOOL     :NAME       symbol
             :VERSION    string
             :FORMATS    (format0, format1, ..., formatn)
             :METHODS    (method0, method1, ..., methodn))
```

where symbol is the name of a tool and string the tool's version number. *Format*$_0$, *format*$_1$, ..., *format*$_n$ are the formats the tool uses for reading and writing files. *Method*$_0$, *method*$_1$, ..., *method*$_n$ are the names of the modeling methods supported by the tool.

3.1.4 Methods

Defining the methods that are available at a specific site is accomplished in a fashion similar to the user role definition. At this point, only the declaration that a method exists and is accepted for use within the system development process is necessary. The form for specifying this information is:

```
(DEFMETHOD   :NAME symbol)
```

where *symbol* is the name of a method. For example, to establish IDEF0 as an accepted method simply involves the specification of the following form:

```
(DEFMETHOD :NAME IDEF0)
```

After defining the methods that are allowed, the framework designer can now use the method names within facts and constraints.

3.1.5 Relations

The relations are the means by which a framework designer can introduce user-defined relations that can be used in the specification of the framework. Once a relation has been defined, the relations can be used in constraint specifications. To define a user-defined relation, the following form is used:

```
(DEFRELATION :NAME symbol)
```

where *symbol* is the name of the user-defined relation. An example of a user-defined relation is shown below. It should be noted that an open issue concerning the DEFRELATION form is whether the number of arguments to the relation should be specified in the DEFRELATION form. Knowing the number of places in the user-defined relation would assist the Framework Processor in performing consistency checking. However, it is unclear at this point if the relations involved would always have the same number of arguments. For this reason, the NUMBER-OF-ARGUMENTS slot, as it would be called, is not currently part of the DEFRELATION form specification.

```
(DEFRELATION  :NAME design-paradigm)
```

The design-paradigm relation can now be used in constraints to better represent an organization's development process. The following example shows how this can be accomplished. Refer to Section 3.2.1.2.3 for a full explanation of the syntax and semantics of this example.

```
(->     (DESIGN-PARADIGM object-oriented)
        (ARTIFACT-ACCESS design-document :METHOD (IDEF4)))
```

If this constraint were included as part of a task's elaboration (see Section 3.2.1.2.3), it would specify that the design-document must be prepared using IDEF4. This example assumes that the (design-paradigm object-oriented) relation was asserted to be true previously. This allows the system development process to dynamically modified based on decisions made during that process.

3.1.6 Users

Often, in the framework definition, the granularity of user roles are not sufficient for specifying access privileges. In these situations, it becomes necessary to refer to a specific user, as in the case where a task can only be signed-off (i.e., completed) by a single person. To define a user, and therefore make that person accessible to the framework definition, the following form is used:

```
(DEFUSER    :NAME       symbol
            :PASSWORD   symbol
            :PROJECTS   (proj-role0, proj-role1, ... , proj-rolen))
```

where the first *symbol* is the user-id of the user, the second *symbol* is the password for logging into the system, and *proj-role*$_0$, *proj-role*$_1$, ... , *proj-role*$_n$ is a set of project-role pairs. An example entry for the user profile data might look like the following:

```
(DEFUSER    :NAME       John-Doe
            :PASSWORD   ******
            :PROJECTS   (<XYZ, Analyst> <ABC, Project Leader>)
```

With definitions of this type, the framework has the knowledge about users that is needed to provide proper access control. For example, suppose the framework specifies a constraint that an artifact for project XYZ can only be accessed by the Project Leader. If John-Doe (above) tried to modify the artifact, the Framework Processor would check the user profile for John-Doe and find that he has the Analyst user role for this project and thus does not have the required authorization to access this artifact.

3.1.7 Projects

The definition of a project serves to provide a placeholder for a development effort being managed by the Framework Processor. This placeholder can serve to partition the information in the knowledge bases by associating facts and constraints with specific projects. This partitioning will eliminate conflicts that would arise from multiple projects using the same framework by allowing the Constraint Propagator to make deductions based only on the assertions surrounding a specific project. Additionally, the project can serve as the destination of messages sent by the Integration Platform to the Framework Processor detailing events surrounding that particular project. Finally, the project definition provides a means of associating a user with a specific project so that the Framework Processor can determine user access privileges for each project. To define a project, the following form is used:

```
(DEFPROJECT        :NAME symbol)
```

where *symbol* is the name of the project. To define the FPP project, for example, would require the following form:

```
(DEFPROJECT        :NAME FPP)
```

The DEFPROJECT form and the other definitional forms are concise in nature. Nevertheless, these forms provide improvements in the operation of the Framework Processor by simplifying the validation process. These statements define the appropriate set of terminology to the Framework Processor so that elements that do not have a definition can be detected as an error in the framework definition.

3.2 Process Specification Component

With an understanding of the terminology and agents (provided by the Definition Component) that will be part of the framework definition, the specification of the more complex development process information can begin. The Definition Component defined the objects produced by the development process and the agents and mechanisms (e.g., tools and methods) used for creating and evolving the objects. With the Process Specification Component to be described in this section, the framework designer can complete the framework definition by specifying where, when, and how those objects should be manipulated (i.e., the place in the system development process where the objects should be created, the tools and methods for creating the objects, and the sequence in which the objects should be created) and who should do the manipulation.

The Process Specification Component of the framework is intended to capture this complex development information by providing a rich set of language constructs for expressing and capturing the complex relationships that exist within development situations. In deriving these language constructs, effort was made to ensure that they be computable. It is with these computable forms that the Framework Processor will manage and control the development process.

The language constructs devised follow the conceptual framework description provided in Section 2.1. In that discussion, the framework was partitioned into two separate views, the Situation Classification Framework and the System Development Framework. While the two views have different areas of focus, there is a considerable amount of overlap in the two views. In fact, because of the expressive power of the process description language, it is possible to completely absorb the information represented in the Situation Classification Framework into the System Development Framework. Figure 11 displays this concept. All development process knowledge is provide by the process descriptions. An external view of a subset of this information can then be defined that represents the classification of the situations that occur during the development process.

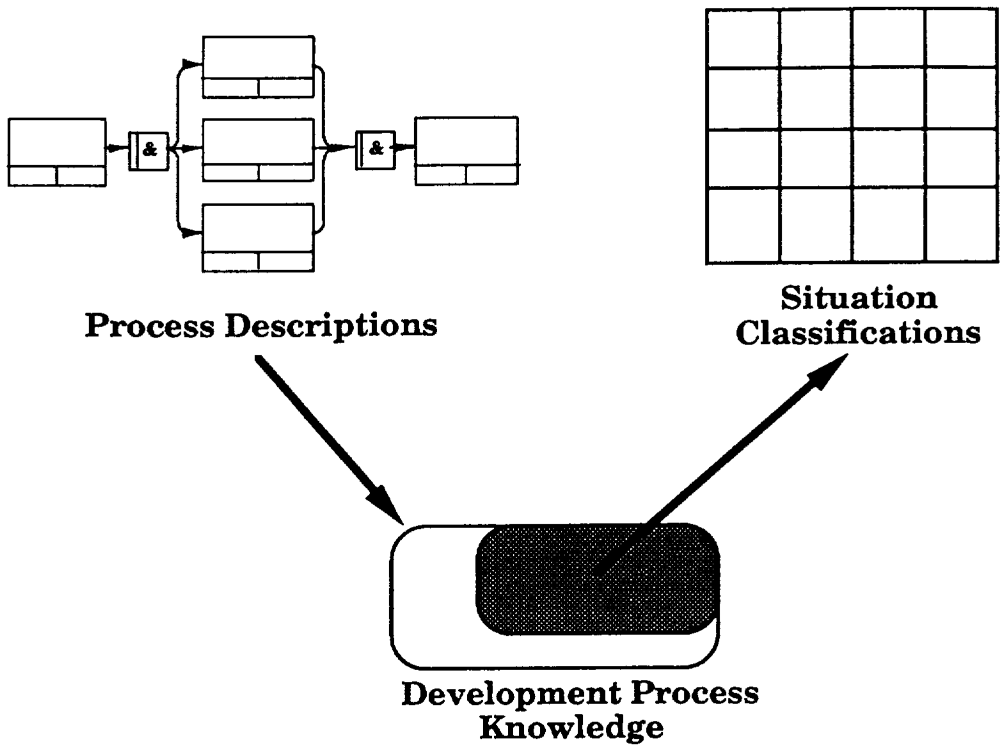


Figure 11. Development Framework Representation

This is precisely the strategy taken by the Process Specification Component. Within the representation structure provided by the Process Specification Component, two modes of information capture are provided. The Process Flow Description structures capture the process oriented nature of the development process. Embedded in this process information is the situation information (e.g., tools, methods, users, and artifacts). Once these descriptions have been defined, a Situation Classification Matrix can be defined to create an external view on the process descriptions that take a more situation oriented focus. The following two sections will describe how the Process Flow Descriptions and Situation Classification Matrix are defined. The first subsection addresses the syntax and structure for the process description diagrams. The second subsection will address the definition of situation classification matrices and how they are derived from the process descriptions.

3.2.1 Process Flow Descriptions

With the Process Flow Descriptions, the framework designer captures the system development process, including the facts and constraints pertaining to that process. The means by which these process descriptions are captured is a set of diagrams based upon the IDEF3 Process Description Capture Method [Mayer 90].

A process, in general, involves objects with certain properties standing in specified relations. A process can also stand in relations with other processes (e.g., a process can start, suspend, and terminate other processes; objects or information about objects can be shared between processes; one process can change the properties of such a shared object and "cause" the exclusion of another process execution; etc). IDEF3 was chosen as the representation scheme for the development process framework because of its power in representing processes. The basic strategy of IDEF3 is centered around the capture of descriptions of process flow (processes and their temporal, causal, and logical relations) in addition to the identification of objects that participate in these processes.

To be consistent with the format of this document, a discussion of the syntax by which IDEF3 diagrams would be presented to the Framework Processor is required. However, a design assumption made for the Framework Processor requires that an IDEF3 tool be tightly coupled with the Framework Processor. This design assumption will be handled in one of two ways: (1) the IDEF3 tool will be subsumed by the Framework Processor or (2) a structured interface between the IDEF3 tool and the Framework Processor will be defined. In either case, the Framework Processor will still have access to the internal representation of the IDEF3 diagrams. For this reason, no textual representation for IDEF3 was produced as part of this design (see Section 5.3 for further discussion on the generation of a textual IDEF3 format).

Despite this lack of formal textual syntax for the IDEF3 diagrams, a discussion of how the diagrams will be interpreted is still possible. However, before turning to the subject of interpretation, a brief overview of the elements of IDEF3 is required.

3.2.1.1 IDEF3 Overview

An IDEF3 Process Flow Description captures a network of relations between activities. An IDEF3 Process Flow Diagram consists, in part, of the following structures:

1) Units of Behavior (UOBs),
2) Elaborations,
3) Junctions, and
4) Links.

An example IDEF3 diagram is shown in Figure 12.

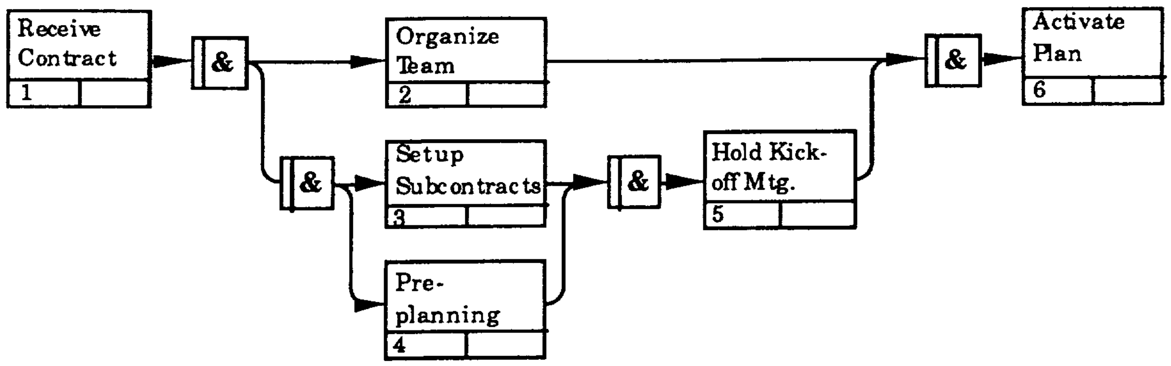


Figure 12. An Example Process Flow Diagram

The Unit of Behavior (UOB) is the basic unit of the Process Flow Diagram and is used to represent a task or activity. In the example, the Receive Contract element represents a UOB. A UOB is displayed with its label, node number, and optional IDEFØ activity reference number. The label should be verb-based to provide some indication as to what process is being represented by that particular UOB. Perhaps the most powerful aspect of the UOB is its ability to be decomposed. Within a decomposition, greater detail as to how a process or activity is performed can be given in the form of another IDEF3 diagram. In essence, a UOB can be described in terms of other UOBs.

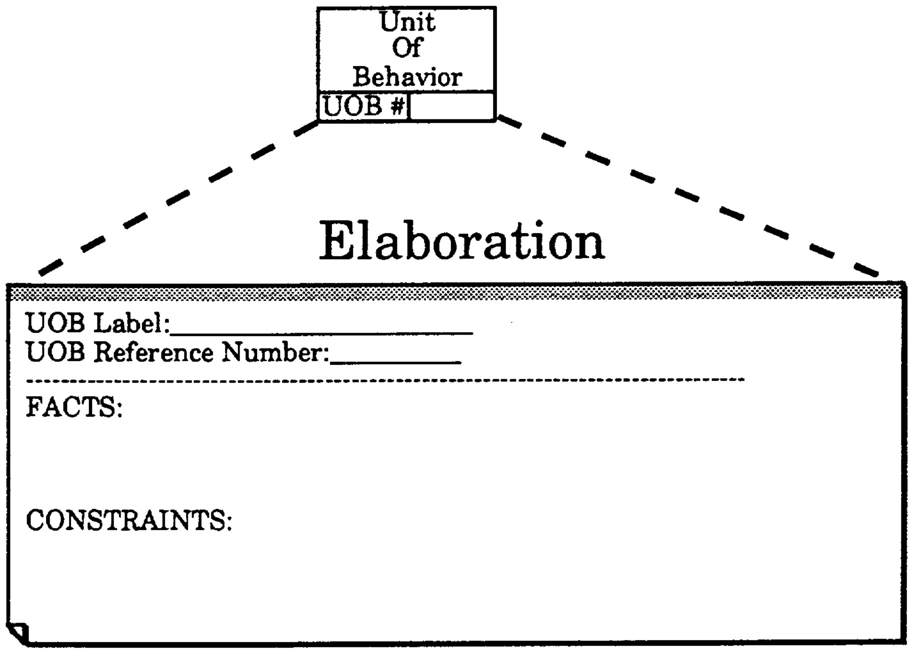


Figure 13. An Elaboration of a UOB.

Another way to capture information about a UOB is through the Elaboration. Every UOB in an IDEF3 diagram can have an Elaboration attached to it. Figure 13 shows a conceptualization of how an Elaboration relates to a UOB. The basic idea is that an Elaboration is simply a form where facts about the UOB and constraints on the UOB can be described. IDEF3 currently places no limitations on the form that these Elaboration statements can take (i.e., natural language). However, the Framework Processor has specified a syntax for the Elaboration specifications (see Section 3.2.1.2.3).

UOBs, in and of themselves, do not sufficiently capture a description of complex processes. To remedy this, IDEF3 provides the mechanisms for arranging these UOBs into complex networks of activities. A relationship between UOBs is represented by a Link. In the example shown in Figure 12, Links are represented by the directed arrows. Though not displayed in the example, IDEF3 has defined three different link types as shown in Figure 14. The Relational link represents a user specified relationship (this should not be confused with user-specified relations defined with the DEFRELATION form); the Precedence link represents a temporal relationship between two UOBs; and the Object Flow link, along with the precedence relationship, specifies that objects participating in the UOB at the source of the link are passed to the UOB at the destination of the link.

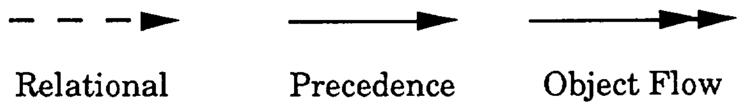
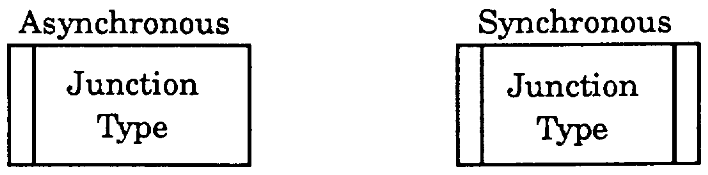


Figure 14. IDEF3 Link Types

To highlight constraints on possible sequencing relations among UOBs, Junctions are used. A Junction can be used to link branches of the process flow that can proceed independently of each other. The representation for a Junction is shown in Figure 15. Each junction has an associated type and sequencing interpretation. The types supported in IDEF3 are AND, OR, and XOR; the semantics of which are equivalent to their logical meanings. Additionally, a Junction can capture relative timing constraints between the different branches by specifying whether the initiation of the branches occurs synchronously or asynchronously.



Junction Type	
&	AND
O	OR
X	XOR

Figure 15. IDEF3 Junctions

3.2.1.2 Framework Definition in IDEF3

The IDEF3 components just described provide a rich set of language constructs that allow a framework designer to provide very detailed specifications of the development process. The use of IDEF3 is made very attractive by the fact that the formal semantics of IDEF3 make IDEF3 diagrams *computable*. However, the desire to maintain the framework definition in an *interpretable* form, within the context of software development processes, has forced the Framework Processor design to place a certain amount of structure on the form the IDEF3 diagrams may take.

This is not to say that IDEF3 was modified to meet the needs of the Framework Processor. Instead, this means that the Framework Processor will require certain interpretations to be made of and certain structures to exist within the IDEF3 diagrams that are part of the framework definition. Each of these interpretations and structures have been added within the syntax of IDEF3. Think of these conventions as analogous to a company's programming standard for code development and documentation. With the programming standard, the company forces a certain structure on the form source code can take, but the structure falls within the syntax rules of the programming language. In a like manner, the structures to be discussed in the following sections fall within the syntax of IDEF3, but makes the operation of the Framework Process less complex.

3.2.1.2.1 Simplifying Assumptions

While it was previously stated that no modifications have been made to the IDEF3 method, certain changes in the interpretation of IDEF3 constructs have been made. The semantics of IDEF3 diagrams have been defined in a formalization of the IDEF3 method [Menzel 91]. For the most part, the Framework Processor will adhere to the interpretations originally prescribed for IDEF3 diagrams. However, in examining the informational needs of the Framework Processor, it was noted that IDEF3 diagrams can become too specific. The diagrams actually provide more information than the Framework Processor is designed to handle.

Asynchronous versus Synchronous Junctions

Referring back to Figure 15, notice that IDEF3 Junctions can be denoted as either Synchronous of Asynchronous. The purpose of this distinction is to detail the temporal relationship that exists between the process branches attached to the junction. A Synchronous junction specifies that the branches should be initiated (for a fan-out junction) or terminated (for a fan-in junction) at the same time while an Asynchronous junction specifies that the branches can be initiated (or terminated) in any sequence.

Within the context of the Framework Processor, there is no distinction between a Synchronous and an Asynchronous junction. This relaxation in the interpretation of the junctions results from the Framework Processor only being involved in monitoring the progression of a system development process. Consequently, at a fan-out junction the Framework Processor is concerned with which branches should be activated and not however whether they should be activated synchronously or asynchronously. This information is encoded in the constraints within the elaboration of the fan-out junctions. Similarly, the fan-in junctions signal the completion of a set of branches based on the fan-in junction type without regard to whether they completed synchronously or asynchronously.

Link Types

Now, referring back to Figure 14, notice the three different link types defined by IDEF3. The Precedence link simply specifies a temporal relationship between two IDEF3 elements (UOBs or Junctions). The Object Flow link carries the same precedence relationship as the Precedence link, but also specifies that specific objects move from one process to the next. The Relation link represents some user-specified relationship between two processes. In the present design, the Framework Processor is only capable of handling precedence relationships. As a result, all links in the IDEF3 diagrams will be interpreted as Precedence links.

It should be noted that despite the fact that the Framework Processor does not distinguish between the junction and link types, the IDEF3 diagrams should still be prepared as if the Framework Process did actually distinguish between them. Most importantly, doing this would provide a more accurate description of the development process in the framework definition. But also, distinguishing between the junction and link types during the initial framework definition would not require modification to the framework when the Framework Processor is enhanced to distinguish between the junctions and the links.

3.2.1.2.1 Atomic versus Compound UOBs

A challenge with using IDEF3 to capture the development process centers around the granularity of the IDEF3 process descriptions. Since IDEF3 supports decomposition of processes, it can be difficult to determine which UOBs represent tasks that can be performed by members of the project team and which UOBs are part of the description as an organizing element. In some situations, the decomposition of a UOB may simply be a more detailed description of a specific task to be performed by an individual while in other situations the decomposition may represent several individual tasks to be performed by several people.

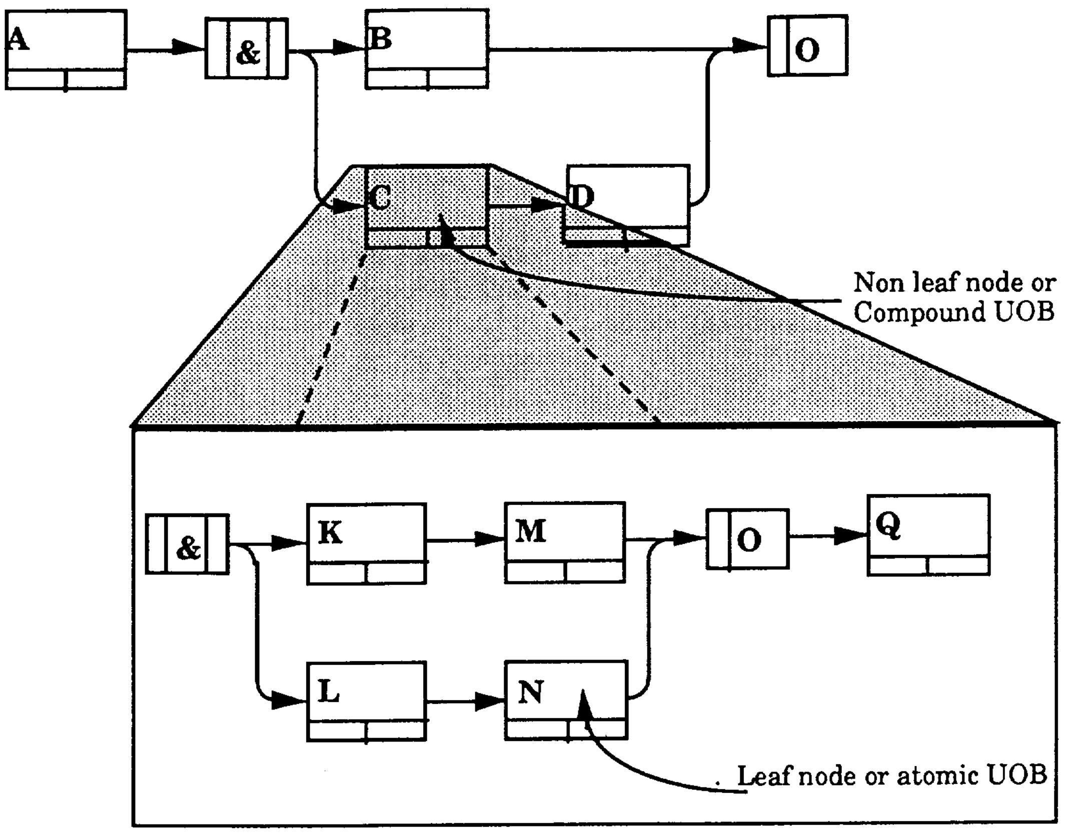


Figure 16. Atomic and Compound UOBs

To address these different situations, the Framework Processor will distinguish between atomic and compound UOBs. Figure 16 illustrates the relationship between these two types of UOBs. Compound UOBs are UOBs for which there is a decomposition. Atomic UOBs are the lowest level UOBs for which there is no decomposition. It is within these lowest, most detailed UOBs that information about the actual task is to be represented. Every one of these leaf UOBs must have an Elaboration Specification attached to specify the access control and completion criteria for the task (see Section 3.2.1.2.3).

However, to address the situation where the decomposition represents a detailed description of an individual task, the constraints specified on the atomic UOBs in the decomposition can be rolled up to the parent compound UOB. This rolling up of constraints could then allow the compound UOB to be treated as an indivisible task in certain development situations where the details of the decomposition are not required to complete the specified task. The ability to roll up the constraints can also apply to the situation where the decomposition represents several individual tasks. For example, referring back to Figure 16, suppose that *programmers* have access to UOBs M, N, and Q. Further, let *analysts* have access to K, L, M, and N and *testers* have access to only Q. By rolling up all allowable access to the parent UOB, C, the Framework Processor can determine that only programmers, analysts, and testers may have access to the C process.

3.2.1.2.3 IDEF3 Elaboration Specification

To every Atomic UOB and Fan-out OR or XOR Junction, an Elaboration Specification must be attached. The visualization for the Elaboration was shown in Figure 13. With the Elaboration, the framework designer must specify the facts and constraints that are important to that task (UOB) or decision (Junction). To follow are the forms by which facts and constraints will be expressed to the Framework Processor. The reader is referred to Appendix B for a description of the grammar for the Elaboration specification.

Facts

The Facts specified in an Elaboration for a task are defined to specify the access control policies for the task represented by the UOB and to identify any artifacts manipulated as part of the task. The general form for specifying access control policies is:

(ACCESS-ROLE *role-type* [:ALL | :EXCEPT *list* | :ONLY *list*])

where *role-type* is the name of a user role and both *list*s are a set of individuals that are members of the user role type. Both the user role type and the users in the list must be defined in the Definition component of the framework specification. This form describes the range of individuals who may be allowed to access this UOB. The :ALL keyword allows anyone to access the UOB. The :EXCEPT keyword allows the framework creator to exclude individuals from the allowable group. The :ONLY keyword allows only the listed users access.

The following are a few examples of the access-role form and their interpretations:

(ACCESS-ROLE programmer :ALL)	All programmers have access.
(ACCESS-ROLE programmer :EXCEPT (tom))	All programmers except tom.
(ACCESS-ROLE programmer :ONLY (tom joe))	Only the programmers tom and joe.
(ACCESS-ROLE :ALL :ALL)	Everyone.

For leaf nodes, there *must be at least one*, and possibly more, of these access-role facts.

Artifact access facts are also included within the facts section of the Elaboration. These facts are used to specify what artifacts are to be used/generated/modified/manipulated within the individual process nodes. The general form for artifact access facts has the structure:

```
(ACCESS-ARTIFACT name   :TOOLS list)
                        :METHODS list
                        :STATE state-of-artifact)
```

where *name* represents the artifact in question, the *list* associated with the :TOOLS keyword represents the set of tools that can be used to manipulate the artifact, the *list* associated with the :METHODS keyword represents the set of methods that should be used in manipulating the artifact, and *state-of-artifact* represents the state the desired artifact must be in before manipulation during this activity. This state information will help insure that the proper version of the artifact is used. As with the ACCESS-ROLE form, elements of this form must have been defined in the Definition Component. The artifact must be valid. The tools list and the methods list must be subsets of the set of tools and the set of methods, respectively, registered to for this artifact. Finally, the state must be one of the possible states defined for the artifact.

The following is an example of the access-artifact form:

```
(ACCESS-ARTIFACT  err-handler-source-code
                  :TOOLS (emacs )
                  :METHODS nil
                  :STATE initial)
```

This form states that the artifact to be accessed is the err-handler-source-code artifact, in its initial state. The form also states that emacs should be used to edit the artifact and no particular method is required.

Constraints

While the facts defined in the Elaboration define more of the situational information about a task, constraints focus more on the dynamic, procedural information about a task. The constraints are used to define the means by which a process is completed and can specify special constraints that must be satisfied during the performance of a task. For a detailed discussion of the relation types that can be used in the constraint statements, the reader is directed to Section 4.1.2. The remainder of this section will be directed at discussing the broad categories of constraints that will be found in the Elaboration specifications.

There are three general categories of constraints found in the framework definition: (1) completion criteria constraints, (2) selection constraints, and (3) general constraints. Completion criteria and general constraints are used to assist in the definition of UOBs while selection constraints capture the decision making logic of Junctions.

Completion Criteria

The completion criteria is defined to establish what requirements must be met in order for an activity to be considered complete. The mechanism for defining a completion criteria is to simply define a relation, as shown below:

```
(ARTIFACT design-document in-review)
```

This example states that the artifact x must reach the in-review state in order for the task to complete. The stated relation can be any user-defined or system defined relation, but for every atomic UOB, there *must be one and only one* completion criteria statement in the Elaboration specification. The previous example is somewhat trivial in that rarely will the completion criteria be just a single relation. However, by using the logical relations AND, OR, or XOR, more complex relations can be specified. For example:

```
(AND (ARTIFACT design-document in-review)
     (USER-SIGNOFF John-Doe))
```

This completion criteria states that the artifact x must be in review and that the user John-Doe has signed off activity. The user-signoff relation specifies that doej must give approval before this task can be considered complete. More than likely, this will be a situation that occurs frequently in the development process.

Selection Criteria

Whereas completion criteria are associated with UOBs, the selection criteria are defined in the elaborations of fan-out junctions. The selection criteria specifies the constraints that will determine which branches of the process description will be activated. Since an AND junction specifies that all branches will be activated, selection criteria must be specified for only OR or XOR junctions. The form of the selection criteria will be a constraint based on either the USER-SELECT or SELECT relations (see Section 4.1.2.2).

General Constraints

General constraints are supported to allow the framework designer to code more complex situations into the framework definition. Through the use of user-defined relations (see Section 4.1.2.3), the Framework Processor can maintain and enforce a set of constraints that are much more detailed than that specified by the IDEF3 diagrams. The IDEF3 diagrams are translated into constraints that enforce the sequencing of tasks or activities. The general constraints can be used to enforce more than sequencing constraints on the development process.

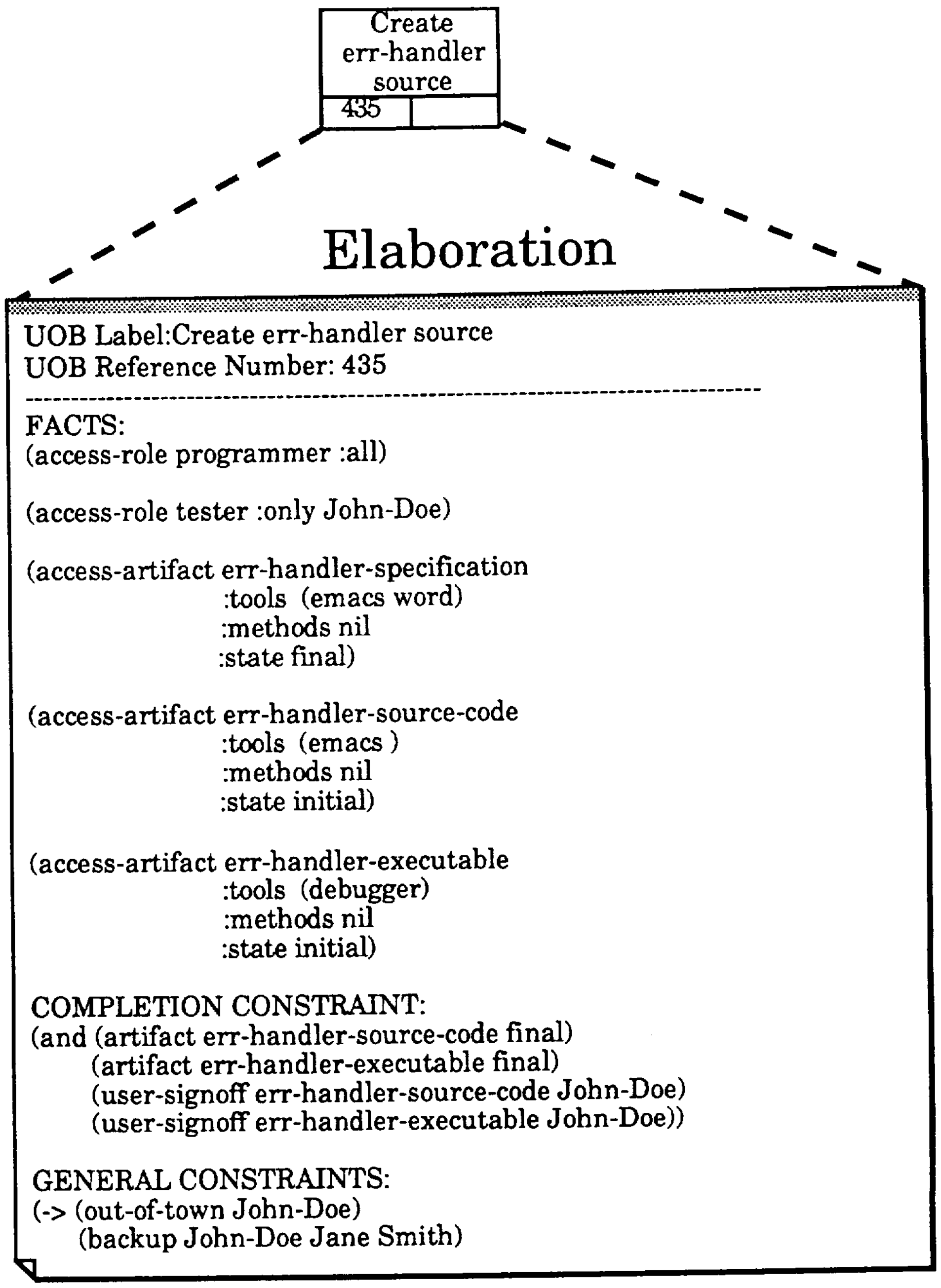
The following example should give insight as to how general constraints could be used. Assume that the UOB for which the constraint is being written is responsible for producing the system design document. The following constraints could specify how that document is to be prepared.

```
(->   (DESIGN-PARADIGM object-oriented)
      (ARTIFACT-ACCESS design-document :METHOD (IDEF4)))

(->   (DESIGN-PARADIGM structured)
      (ARTIFACT-ACCESS design-document :METHOD (DFD)))
```

The first constraint implies that, if at some time in the past, the design-paradigm for this project was established to be object-oriented, then IDEF4 should be used to prepare the system design. If, however, a more structured system design approach is chosen, the design should be prepared with Data Flow Diagrams (DFD). So, this example demonstrates that, though the framework is general enough to capture the overall development process, the framework can be specialized to address certain characteristics of specific types of projects. This specialization is achieved through the specification of general constraints. The reader is directed to Section 4.1.2 for more on specifying constraints.

Figure 17 illustrates a sample UOB elaboration specification for the CREATE-ERR-HANDLER UOB. In the facts section, the first access statement specifies that all *programmers* are allowed access to this process. The second statement specifies that only the user *John-Doe* serving in the role of *tester* is allowed to participate in this activity. The next three facts are artifact related and describe what artifacts are manipulated during this process. Following the facts section of the elaboration is the constraints section. As mentioned, each UOB elaboration must contain one completion condition and the constraint specified in this example is one such constraint. This constraint states that the artifacts *err-handler-source-code* and *err-handler-executable* must reach their *final* state and the user *John-Doe* must SIGN-OFF on these two artifacts in order for the CREATE-ERR-HANDLER-SOURCE task to be completed. Finally, the elaboration can contain general constraints. In this case, the constraint specifies that if John-Doe is out-of-town, then Jane-Smith is the backup for John-Doe. While this diagram relates the Elaboration to a UOB, the Elaboration for a Junction would be very similar.



```
UOB Label:Create err-handler source
UOB Reference Number: 435
---------------------------------------------------------------------------
FACTS:
(access-role programmer :all)

(access-role tester :only John-Doe)

(access-artifact err-handler-specification
                 :tools (emacs word)
                 :methods nil
                 :state final)

(access-artifact err-handler-source-code
                 :tools (emacs )
                 :methods nil
                 :state initial)

(access-artifact err-handler-executable
                 :tools (debugger)
                 :methods nil
                 :state initial)

COMPLETION CONSTRAINT:
(and (artifact err-handler-source-code final)
     (artifact err-handler-executable final)
     (user-signoff err-handler-source-code John-Doe)
     (user-signoff err-handler-executable John-Doe))

GENERAL CONSTRAINTS:
(-> (out-of-town John-Doe)
    (backup John-Doe Jane Smith)
```

Figure 17. Example Elaboration

3.2.2 Situation Classification Matrix

The Situation Classification Matrix is a visualization mechanism for a subset of the information represented in the development framework. The framework designer must have the ability to define what the structure of this matrix will be. This is accomplished using the following form:

```
(DEFMATRIX   :ROWS list-of-strings
             :COLUMNS list-of-strings
             :CONTENTS list-of-situations
```

where the first *list-of-strings* represents the names of the rows of the matrix, the second *list-of-strings* represents the names of the columns of the matrix, and *list-of-situations* represents the contents of the cells of the matrix. Each of the situations in this list must match the name of a UOB in the top level IDEF3 diagram of the process component of the framework. It will be through these names that a connection between a cell in the matrix and a process in the process descriptions will be made.

The following form provides an example and describes the matrix shown previously in Figure 10, with the addition of row and column names:

```
(DEFMATRIX   :ROWS      (  "objectives" "domain" "business"
                           "information" "technology")
             :COLUMNS   ( "data" "user" "function" "network"
                          "col5" "col6")
             :CONTENTS  (   A    nil   B    C    D    E
                            I    F    nil  nil  nil  nil
                            J    G    nil  nil  nil  nil
                           nil   H    nil  nil  nil  nil
                           nil  nil   nil  nil  nil  nil))
```

The resulting chart will be used by the Session Manager (see Section 4.6) and gives the viewer a visual picture of the various situations that exist within the development process. These development situations are displayed in relation to one another with respect to the perspective and focus of each situation. This serves to divide the "world" of software development into pieces and helps the development team concentrate on specific situations. This should alleviate some of the burden of trying to understand the entire software system in order to conceptualize a part of the system.

4 Framework Processing Functionality

In Section 2.2.2, the architecture for the Framework Processor was described at a functional level. The purpose of this section is to detail the operation of the major functional components within this Framework Processor architecture. To facilitate the discussion of the components, this section has been divided into six subsections. The first subsection gives an overview of constraint propagation. Accompanying this overview is a discussion at the conceptual level on how the Framework Processor uses the Constraint Propagator to control the system development process. This discussion also includes a description of the relations used to build the constraints that populate the knowledge base used by the Constraint Propagator. The second subsection describes the three tiered validation procedure applied by the Framework Processor to the framework. This validation procedure is used to ensure the correct operation of the Framework Processor during the control of a system development process. The third subsection describes the process used by the Framework Processor to convert the IDEF3 process descriptions – defined in the framework – to constraints used by the Constraint Propagator. The fourth subsection describes the operation of the Fact Base Manager which is the Framework Processor component responsible for maintaining facts defined in the elaborations of the IDEF3 process descriptions. The fifth subsection describes the operation of the Framework Processor during the control of the system development process. The final subsection centers around the description of the Session Manager component of the Framework processor. The Session Manager controls a direct interface between the framework users and the Framework Processor.

4.1 Constraint Propagation

The term *constraint propagation* refers to a general class of techniques for computing the effect of new information on existing knowledge base. Both the new information and the knowledge base are represented as constraints. Upon receiving a new piece of information, a constraint propagation system makes deductions using the standard rules of predicate logic. This new piece of information may cause contradictions within the knowledge base. At that point, a constraint propagation system determines the original assertions which lead to the contradiction. The constraint propagation system presents the originator of the information with the set of assertion leading to the contradiction. The originator selects the assertion to be retracted (i.e., the piece of information to remove from the known set of knowledge) thereby removing the contradiction from the knowledge base maintained by the constraint propagation system.

The Framework Processor uses the concept of constraint propagation to handle the effects of events during the system development process on the framework. The remainder of this section focuses on describing the Constraint Propagator component of the Framework Processor. The discussion is divided into two parts. The first part presents a description of the interaction between the Constraint Propagator component and the Framework Manager. The second part defines the primitive relations used to build the constraints managed by the Constraint Propagator.

4.1.1 Constraint Propagator Overview

The Framework Processor, via the Framework Manager, feeds the Constraint Propagator new knowledge representing the progress of tasks and artifacts. Using this new knowledge the Constraint Propagator makes inferences based on the process description knowledge encoded in the form of constraints. Figure 18 illustrates this interaction between the Framework Manager and the Constraint Propagator.

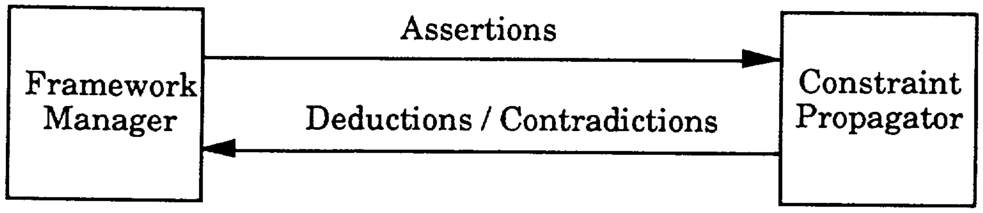


Figure 18. Interaction of the Framework Manager and the Constraint Propagator

Notice the types of information being passed along the lines of communication. The Framework Manager informs the Constraint Propagator about a set of assertions which it receives by way of the Platform Interface Manager from the Integration Platform. These assertions represent changes in the state of the development process. The Constraint Propagator uses these assertions and deduces any new information it can from these assertions. The Constraint Propagator then reports back to the Framework Manager the newly deduced information about the current state of the system development process. If these assertions lead to a contradiction, the Constraint Propagator reports to the Framework Manager that an inconsistent state has been reached. The Framework Manager must then take action to re-establish a consistent state. This action always results in the retraction of some assertion. Therefore, the Constraint Propagator isolates those assertions which lead to the contradiction, and presents them to the Framework Manager. The Framework Manager notifies the framework administrator. The framework administrator selects the appropriate assertion to be retracted. Having been restored to a consistent state, the Constraint Propagator continues processing. In this way, the Constraint Propagator uses information encoded in the form of constraints along with the assertions made by the Integration Platform to monitor and control the system development process.

4.1.2 Constraint Relation Types

The constraint relation syntax used by the Framework Processor is a Lisp-like syntax. These relations are be used to build up complex constraints for expressing development process information in the framework definition. These relations can be divided into three categories. These categories are:

1) logical relations,
2) trigger relations, and
3) user-defined relations.

In the following subsections, each category of constraint relations is discussed by (1) describing the syntax for the relation and (2) explaining the semantics associated with the constraint relation.

4.1.2.1 Logical Relations

The logical relations consist of three connective operators, a negation operator, and a conditional operator. The connective operators have the following form:

(*op* $expr_1$ $expr_2$ ... $expr_n$)

where *op* is one of the connective operators (e.g., AND, OR, and XOR) and $expr_1$, $expr_2$, ..., $expr_n$ are boolean expressions. The connective operators are a boolean value. The AND relation is true if and only if all of the boolean expressions are true. The OR relation is true if at least one of the expressions are true. The XOR relation is true if one and only one expression is true and the remaining are false.

The negation relation has the following form:

(NOT *expr*)

where *expr* is a boolean expression. The negation operator simply toggles the value of the boolean expression, that is, if the boolean expression is true, then the negation relation will return false, and vice versa.

The conditional relation is the same as the implication operator used in predicate logic. The conditional relation has the following form:

(-> $expr_1$ $expr_2$)

where $expr_1$ and $expr_2$ are boolean expressions. $Expr_1$ represents the antecedent, and $expr_2$ represents the consequent. If the antecedent is true, then $expr_2$ is asserted to be true by the rule of inference known as *modus ponens*. If the consequent is false, then the antecedent is asserted to be false by the rule of inference known as *modus tolens*. For the cases where either the antecedent is false or the consequent is true, no additional inferences can be made.

4.1.2.2 Triggering Relations

Five triggering relations have currently been identified. They include START, DONE, SELECT, USER-SELECT, and USER-SIGN-OFF. These five relations along with the logical relations are used to specify the completion criteria for a leaf UOB. The START relation is used by the framework processor to indicate that a task has been started. The START relation has the following form:

(START *A*)

where *A* is a unique identifier representing either a unit of behavior (UOB) or a junction. For the framework processor to determine whether task *A* has been started, it queries the constraint propagator's knowledge base to determine if (START *A*) is true. If so, the task *A* has been started. However, the task may have already been completed. To determine this, the framework processor must determine if (DONE *A*) is true. The DONE relation indicates whether a task has completed and has the same form as the START relation. Table 1 shows how to interpret the resulting values based on a query for both the START and DONE relation for a given task.

Start Relation	Done Relation	Interpretation
nil or false	N/A	task never started
true	nil or false	task started, but has not finished
true	true	task has started and completed

Table 1. Interpretation of Start/Done Relations

The SELECT relation allows the framework designer to indicate which of the competing fan-out branches to actually perform in the case of a fan-out OR or XOR junction. For example, assume that only task A_1 should be performed under some certain conditions (see Figure 18), then one of the constraints in the elaboration for the fan-out OR junction would specify:

(-> *expr* (SELECT A_1))

where *expr* is a boolean expression representing the condition that must be met for task A_1 to be selected, that is, for task A_1 to be initiated. This example is some what simplified in that only one branch is selected. In other situation multiple branches could be selected, resulting in more complex forms.

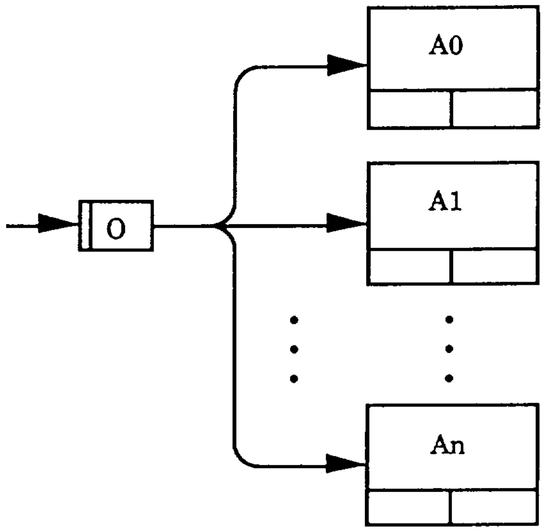


Figure 19. Example of a Select Relation

The SELECT relation allows the framework designer to statically encode which branch of a fan-out OR or XOR junction to activate. The USER-SELECT relation provides a means for the framework designer to specify that the selection of the tasks to activate at the fan-out OR or XOR junction is to be specified dynamically at run-time (i.e., to be specified during the actual controlling of a system development process). The USER-SELECT relation has the following form:

(USER-SELECT *min max* $const_0$ $const_1$ ... $const_n$)

The *min* and *max* indicate the number of tasks that can be selected. For an XOR fan-out junction *min* and *max* can only be one. $Const_0$, $const_1$, ..., $const_n$ are additional constraints used to specify any interrelationships that might exist between different combinations of task selections. These are used to constrain the framework user in the selection of acceptable branch combinations. Using the diagram in Figure 19, if the framework designer wanted to encode the fact that one or two branches should be pursued at the fan-out junction with the additional constraint that if task A_1 is selected then task A_2 should not be selected, the following USER-SELECT relation would be specified:

(USER-SELECT 1 2 (AND (SELECT A_1) (NOT (SELECT A_2))))

The USER-SIGN-OFF relation is used to signify those situations were an individual must give his/her approval before a task is considered completed. The form of this relation is:

(USER-SIGN-OFF $access_0$ $access_1$ ... $access_n$)

where *access*$_0$, *access*$_1$, ..., *access*$_n$ is either a user role (e.g., designer, programmer, etc.) or a specific individual. Any individual with the appropriate access privileges – based on the information in the USER-SIGN-OFF RELATION – can sign-off on the given task.

4.1.2.3 User-defined Relations

The Framework Processor provides a means for expressing user-defined relations. The user-defined relations allow the framework designer to define new relations that are needed to encode a particular system development process. The user-defined relations are atomic in nature, that is, the user-defined relations as a whole can be asserted to be true or false. In contrast to the logical relations and the trigger relations, the user-defined relations do not elicit any special behavior from the Constraint Propagator. The user-defined relations have the form:

(*rel val*$_0$ *val*$_1$ ... *val*$_n$)

where *rel* is the name of a user-defined relation and *val*$_0$, *val*$_1$, ..., *val*$_n$ are the values associated with this relation.

Using user-defined relations, the framework designer can develop a process description that can "configure" itself. That is, the decisions made during tasks early in the development process can effect how the later tasks are accomplished. For example, if one of the initial tasks is to define the software paradigm to use, then the design documents produced – for the software during later tasks – must use the appropriate design paradigm. To encode this information, the framework designer could introduce a user-defined relations of the form:

(DESIGN-PARADIGM *par*)

where *par* is the name of a design paradigm. The task assigned to the job of selecting the design paradigm could then assert:

(DESIGN-PARADIGM object-oriented)

As part of the elaboration of the task to produce the design documents, the following form would appear:

```
(->    (DESIGN-PARADIGM object-oriented)
       (ARTIFACT-ACCESS design-document :METHOD (IDEF4)))
```

Thus, the task assigned to selecting the design paradigm has affected tasks performed later in the development process. In this way, the framework designer can encode the dynamic relationships between tasks.

4.2 Framework Validation

In order to ensure the proper operation of the Framework Processor and its Constraint Propagator component, a validation test must be performed on the IDEF3 description representing the site specific system development process framework. During this validation process, the Framework Processor attempts to verify that the constraints represented by the process descriptions will not produce a contradiction in the Constraint Propagator. Contradictions arise from two sources: (1) invalid IDEF3 syntax or (2) invalid/conflicting elaboration constraints.

The validation process performs the same operation as a compiler. First, the Framework Processor parses the IDEF3 description to determine whether the box and arrow part of the IDEF3 description conforms to the syntax rules specified in the IDEF3 formalization [Menzel 91]. If syntax violations are encountered, the validation procedure will guide the framework designer in resolving these violations. Second, the Framework Processor converts the box and arrow part of the IDEF3 description into a collection of constraints. These constraints, along with any additional constraints specified in the elaborations of UOBs or junctions, are loaded into knowledge bases maintained by the Framework Processor. At this point, the Constraint Propagator determines whether any contradictions have arisen during this conversion process. If a contradiction exists, the Constraint Propagator determines the set of constraints which produced the contradiction. From this set of constraints, the framework designer selects the constraint to be retracted. Upon retraction, the Constraint Propagator checks again to determine whether any contradictions exist in the reduced set of constraints. This process continues until all contradictions have been eliminated.

Although the IDEF3 description has been 'compiled', it still has not been determined whether the 'program' will function properly. The approach used by the Framework Processor is to simulate the system development process represented by a given framework to determine whether any contradictions occur. However, it is impractical to enumerate all of the execution paths or to analytically verify the IDEF3 description. To overcome this problem, the Framework Processor will simulate only a selected subset of execution paths. Since contradictions can still occur during the actual controlling of a system development process, the Framework Processor must provide an interactive means of dynamically modifying the framework to correct this situation.

4.2.1 IDEF3 Syntax Validation

During the syntax validation phase, the framework processor parses the site specific system development process framework to determine whether the IDEF3 description conforms to the formalization for the IDEF3 method. Although the modeling tool used to produce the description should have its own syntax validation procedure, the framework processor cannot rely on this fact. Therefore, the framework processor must perform its own syntax analysis to verify the conformance of the framework with the established IDEF3 formalization. The following is a partial list of syntax rules which must be enforced during the validation process:

1) Only one prediagram allowed per decomposition. A prediagram is a collection of connected nodes.
2) A scenario and a decomposition can have only one leftmost point. A leftmost point of a prediagram is a UOB or junction that does not have any entering links.
3) A scenario can have multiple rightmost points, but a decomposition can have only one rightmost point. A rightmost point of a prediagram is a UOB or junction that does not have any existing links.
4) Every fan-in junction must have a corresponding fan-out junction.
5) A fan-out OR junction cannot be matched with a fan-in AND junction.
6) A fan-out AND junction cannot be matched with a fan-in XOR junction.
7) A fan-out XOR junction cannot be matched with a fan-in AND junction.
8) A loop back cannot occur in the scope of a fan-in junction. The scope of a fan-in junction is defined to be all of the nodes between the fan-in junction and its matching fan-out junction.

The framework processor will be designed around the IDEF3 formalization contained in [Menzel 91]. Thus, the reader is referred to that technical report for further information on the IDEF3 syntax.

4.2.2 Framework Semantic Validation

The semantic validation phase occurs at two levels. The first level is the instantiation validation, and its purpose is to determine whether any contradictions occur upon converting the IDEF3 description of a framework into constraints. The second level is the simulation validation, and its purpose is to simulate the actual controlling of a system development process to determine whether any run-time contradictions occur. Each of these two semantic validation levels will be described in the following sections.

4.2.2.1 Instantiation Validation

The instantiation validation phase determines whether contradictions have occurred during the conversion of the IDEF3 description to constraints. These contradictions are caused by the constraints specified in the elaborations. The elaboration constraints are free-form, that is, by following a few simple syntax rules, the framework designer can generate any kind of constraint. This flexibility has its advantages and disadvantages. An advantage is the expressive power provided to the framework designer to customize a framework for a particular system development process. A disadvantage is that it allows the framework designer to paint themselves into a corner by specifying inadvertently constraints that contradict each other. The contradictions that are identified during this instantiation validation are of the form:

a
(NOT a)

where a is some well-formed formula built from the constraint relations. Clearly, when the propositions are stated in this fashion, there exists a contradiction. However, two conditions hide the identification of these contradictions from the framework designer. First, a can be any complex formula using any number of connectives (e.g., AND, OR, and, XOR). Second, a and (NOT a) can be specified in any elaboration within a system development process description. It is the job of the instantiation validation to catch these obvious semantic errors.

4.2.2.2 Simulation Validation

The simulation validation is the final phase in the validation process. This phase of validation is analogous to the testing phase in software development. In program development, this phase tends to be a long and time consuming effort. As in software testing, it is impractical to enumerate all of the execution paths, and currently no automatic verification technique exists. For these reasons, the Framework Processor will adopt a simulation approach for this validation phase. The framework processor will simulate a set of different possible execution paths to determine whether any contradictions arise during the simulated control of a system development process.

Contradictions that arise during execution are produced by constraints with the following general form:

(-> f_0 c)
(-> f_1 (NOT c))

where f_0, f_1, and c are well-formed formulas built from the constraint relations. There are two specializations of this form. The first specialization is when f_0 is equivalent to f_1. This specialization leads to the following form:

(-> a c)
(-> a (NOT c))

This specialization of the general form illustrates the fact that during execution if the proposition a is asserted to be true, an immediate contradiction will arise in the Constraint Propagator. This contradiction is caused by *c* and *(not c)* both having the same truth value. This specialization, where f_0 is equivalent to f_1, always indicates a contradiction.

The second specialization is when f_0 is not equivalent to f_1. This specialization leads to the following form:

```
(-> a c)
(-> b (NOT c))
```

In this case, if the proposition *a* and *b* are asserted to be true, a contradiction will arise in the Constraint Propagator. Unlike the first specialization which always indicates an error in the constraint specification, the second specialization does not necessarily indicate a contradiction within the context of the framework. It may be the case that within the given framework it is impossible for *a* and *b* to be true. Therefore, even though these constraints have the potential for producing a contradiction, it does not necessarily indicate that a contradiction will occur.

In summary, the simulation validation phase attempts to find the contradictions that might arise during the actual controlling of a system development process. Because IDEF3 allows loop backs, it is impractical to enumerate all of the execution paths. Since no analytical method exists for verifying the constraints produced by an IDEF3 description are contradiction-free, an alternate method of verification had to be developed. The approach taken by the framework processor during this phase is to simulate a set of execution paths. However, just like software testing, this simulation approach to semantic validation will not be able to find all of the contradictions. Therefore, the Framework Processor must be instilled with the capability of handling contradictions during execution which have eluded detection during the validation process. This capability is needed even if the problem identified above was solved.

4.3 Constraint Generation

This section describes what constraints are produced by the IDEF3 description of a framework. Two classes of constraints are produced during this instantiation phase. The first class of constraints control the activation of successive tasks based upon the completion of their predecessors. These constraints are called trigger constraints and are produced by the links in the IDEF3 description. The second class of constraints encode the completion conditions that must be met in order for a task to be considered completed. These constraints are specified in the elaborations of the UOBs and junctions. Each class of constraints will be presented in more detail in the following sections.

4.3.1 Trigger/Link Constraints

Each link in a diagram represents a trigger constraint. In order to expedite the discussion of converting links into constraints, the links have been divided into three categories: basic, fan-in, and fan-out as shown in Figure 20.

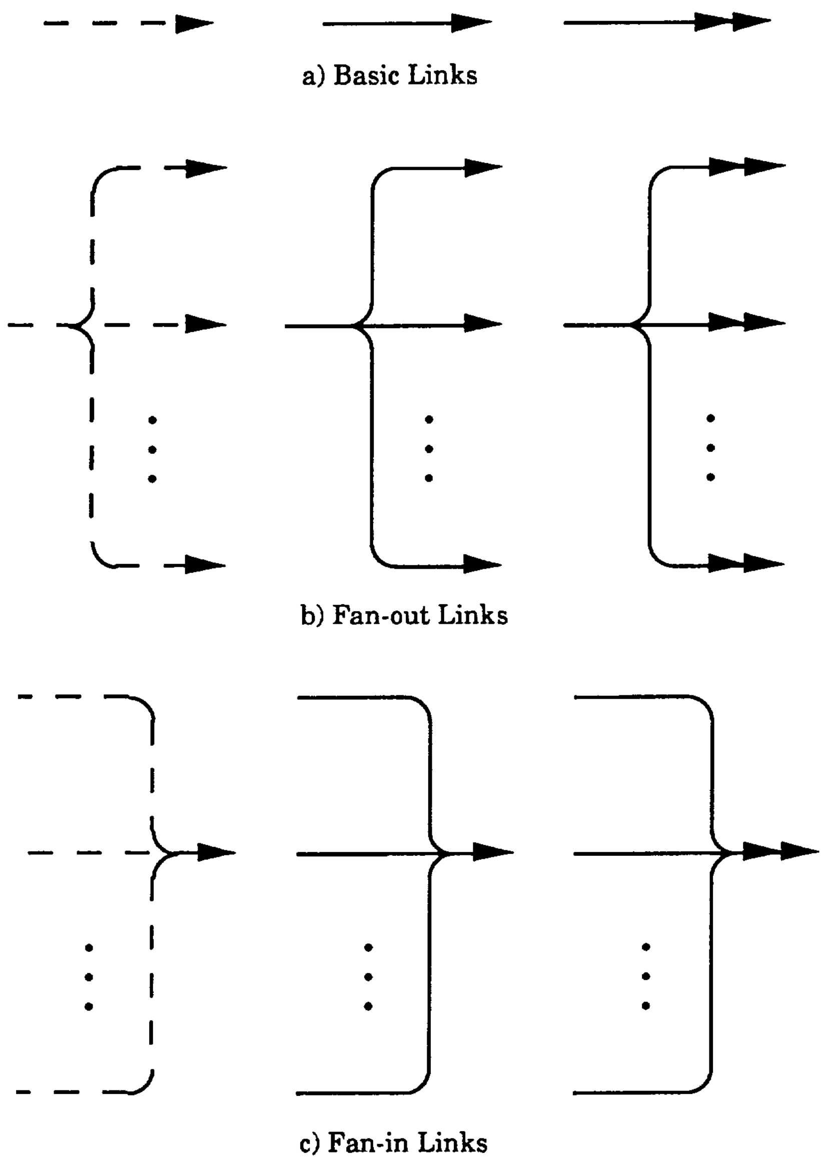


Figure 20. Categories of Links

A link is categorized in this scheme without regard to the type of link (e.g., relational, precedence, and object flow). Each link category produces a different set of constraints and will be described in the following sections.

4.3.1.1 Basic Links

A basic link in a diagram represents a trigger constraint between (1) two UOBs, (2) a UOB and a fan-out junction, or (3) a fan-in junction and a UOB. For example, the completion of the task "Examine Requirements" triggers the start of the task "Decompose Solution" as shown in Figure 21.

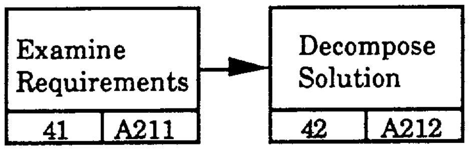


Figure 21. Example of a Basic Link

The trigger constraint produced by this example is:

```
(->   (DONE "Examine Requirements")
      (START "Decompose Solution"))
```

Thus, in the normal case a basic link produces the following constraint:

(-> (DONE A) (START B))

where A is the label of the box that is at the start of the basic link, and B is the label of the box at the end of the basic link. To allow for loop backs in an IDEF3 description, the syntax allows a box to have one or more basic links entering it and no other links entering. A UOB which has multiple basic links entering it is represented by the following general basic link constraint:

(->   (OR (DONE A_0) (DONE A_1) ... (DONE A_n))
      (START B))

where A_0, A_1, ..., A_n are the labels of the boxes that start the basic links ending at the box labeled B. For example, the "Examine Requirements" task in Figure 22 has two basic links entering it.

The trigger constraint for the "Examine Requirements" task is:

```
(->   (OR   (DONE "Requirements Definition")
            (DONE "Solution Limitations"))
      (START "Examine Requirements"))
```

Thus, the completion of either the "Requirements Definition" task or the "Solution Limitations" task signals the activation of the "Start Requirements" task.

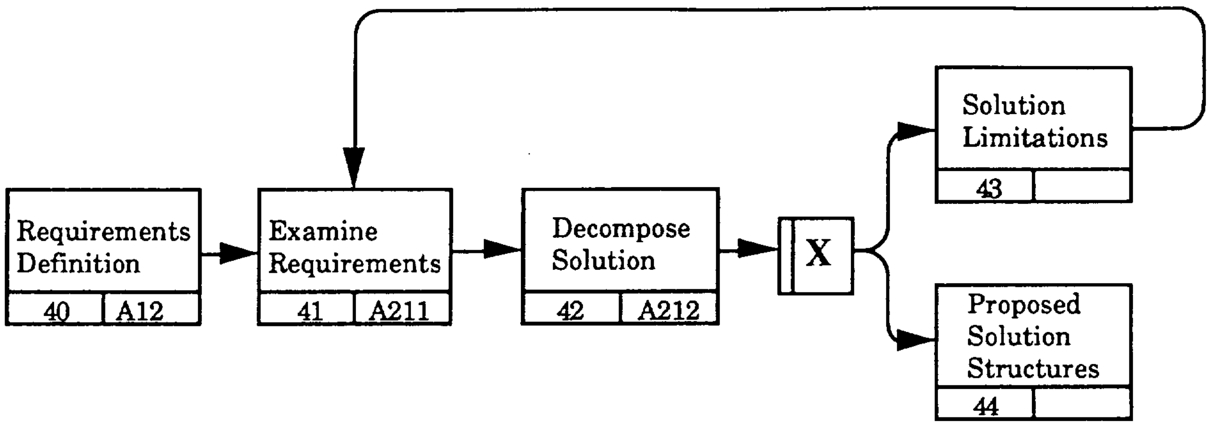


Figure 22. Multiple Entering Basic Links

4.3.1.2 Fan-in Links

A fan-in link represents a relationship between a set of boxes (e.g., UOBs or other fan-in junctions) and a fan-in junction. The constraint generated by the fan-in link depends on the junction type. Whether a junction is synchronous or asynchronous has no bearing on the constraint generated. The general form of the constraint produced by a fan-in link is as follows:

```
(->    (op (DONE A0) (DONE A1) ... (DONE An)))
       (START B)
```

where *op* is the junction type (e.g., AND, OR, and, XOR), A_0, A_1, ..., A_n are the unique names of the boxes at the start of the fan-in link, and B is the unique named associated with the fan-in AND junction. For example, the "Assess Technology Options" task and the "Evaluate Requirements" task must be completed before the fan-in AND junction can be activated as shown in Figure 23.

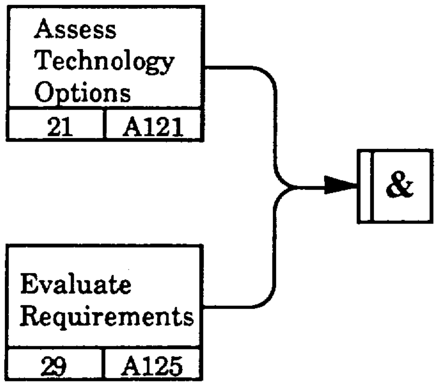


Figure 23. Example of a Fan-in Link

The trigger constraint for this example is:

```
(->    (AND  (DONE "Assess Technology Options")
             (DONE "Evaluate Requirements"))
       (START "AND123"))
```

This example illustrates the fact that each junction in an IDEF3 description must have a unique identifier to allow the framework designer to reference junctions within constraints.

4.3.1.3 Fan-out Links

A fan-out link represents a relationship between a junction and a set of boxes. The fan-out link constraints are generated based on the associated fan-out junction type without regard to whether the junction is synchronous or asynchronous.

The general form of the trigger constraint representing a fan-out AND junction is as follows:

(->   (DONE A)
       (AND (START B_0) (START B_1) ... (START B_n)))

where A is the unique name associated with the fan-out junction and B_0, B_1, ..., B_n are the unique names of the boxes at the end of the fan-out link.

The fan-out OR or XOR junction instantiation is complicated by the fact that something must specify which branch of the fan-out should be pursued. Two techniques for encoding the selection have been provided to the framework designer. One technique allows the framework designer to specify the constraints that must be satisfied for a branch to be selected, and the other technique is simply a form that queries an authorized person for the set of branches to be pursued. The number of branches that may be selected depends on the junction type. The OR junction type allows one or more branches to be selected, and the XOR junction type allows one and only one branch to be selected.

For the OR and XOR junction types, each branch in the fan-out produces the following form:

(->   (AND (DONE A) (SELECT B))
       (START B))

where A is the unique identifier for the fan-out junction, and B is the unique identifier representing the destination of a branch. The XOR junction type produces an additional constraint to restrict the selection of branches to one and only one branch. This additional constraint has the form:

(XOR (SELECT B_0) (SELECT B_1) ... (SELECT B_n))

where B_0, B_1, ..., B_n are the unique identifiers representing the destinations of the fan-out branches.

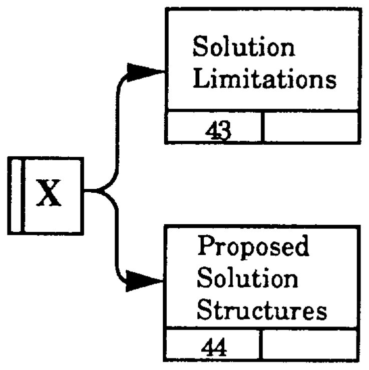


Figure 24. Example of a Fan-out Link

An illustration of a fan-out link originating from a fan-out XOR junction is given in Figure 24. This fan-out link produces two branch constraints plus the additional XOR constraint. These three constraints are:

```
(->    (AND   (DONE "XOR234")
              (SELECT "Solution Limitations"))
       (START "Solution Limitations"))

(->    (AND   (DONE "XOR234")
              (SELECT "Proposed Solution Structures"))
       (START "Proposed Solution Structures"))

(XOR   (SELECT "Solution Limitations")
       (SELECT "Proposed Solution Structures"))
```

These constraints only encoded the triggering information, that is, the information necessary to signal to the next process that its predecessor has finished. The actual selection of the competing branches is encoded in the elaboration of the junction and will be discussed in the next section.

4.3.2 Box constraints

The crux of the Framework Processor in the controlling of the system development process is determining the completion of a task. To signal the completion of a task, the framework designer can choose from one of two techniques. The first and simplest technique is to have the framework user tell the Framework Processor when a task is completed. The second technique is to encode the completion criteria into the constraints in the elaborations and have the Framework Processor, via communication with the Integration Platform, determine automatically upon an artifact's check-in when a task has been completed. This section describes how the elaborations for the UOBs and junctions are converted into constraints maintained by the Constraint Propagator.

4.3.2.1 UOBs

The constraint information for the elaborations of the UOBs have been partitioned into two parts. The first part of the constraint information is used to encode the completion criteria for the task, and the other is a set of general constraints that exist for the task.

Each UOB produces the following completion constraint:

```
(->   (AND   (START A)
             comp)
      (DONE A))
```

where A is the unique identifier for the UOB under consideration and *comp* is the user specified completion criteria for the UOB. Each leaf UOB should have a completion criteria specified for it, otherwise the completion criteria for the UOB simply reduces to the UOB having been started.

A UOB that has an objective decomposition should not specify the completion criteria. Instead two bridge constraints are produced in place of the completion criteria. One is an initiation bridge constraint, and the other is a termination bridge constraint. The initiation bridge constraint has the following form:

(-> (START A) (START a_l))

where A is the unique identifier of the parent UOB and a_l is the unique identifier of the leftmost point in the objective decomposition. This bridge constraint states that the initiation of the parent UOB signals the initiation of the objective decomposition.

The termination bridge constraint works in a similar fashion to the initiation bridge constraint. This constraint indicates that the completion of the objective decomposition process also signals the completion of the parent UOB. Therefore, the general form of the termination bridge constraint is:

(-> (DONE a_r) (DONE A))

where a_r is the unique identifier of the rightmost point in the objective decomposition and A is the unique name of the parent UOB.

4.3.2.2 Junctions

The majority of the logic for the junctions has been encoded by the fan-in or fan-out link associated with a particular junction. The only information that must be specified by the constraints generated by the fan-out OR/XOR junction is the selection of the particular branches to pursue. To facilitate this selection process, these two fan-out junction types have been augmented with an identical elaboration structure to that of a UOB.

All junctions produce the following bridge constraint:

```
(-> (START A) (DONE A))
```

where A is the unique identifier for the junction. This bridge constraints states that the completion criteria for the junction is simply the initiation of that junction.

Since all junctions including the fan-out OR/XOR junctions produce the identical bridge constraints as represented previously, the framework designer is only responsible for specifying the selection criteria in the general constraint section of the junction's elaboration. Each of these constraints will have one of the three following general form:

```
(->   (AND   (START A)
             cond)
      (SELECT A0)

(->   (AND   (START A)
             cond)
      (AND (SELECT A0) (SELECT A1) ... (SELECT An))

(->   (AND   (START A)
             cond)
      (USER-SELECT min max const0 const1 ... constn)
```

The *cond* is some selection criteria for a given branch, A is the unique identifier representing the fan-out junction, A_0, A_1, ..., A_n are the unique identifiers representing the destination boxes of a branch. The *min* and *max* indicate the number of tasks that can be selected. For an XOR fan-out junction *min* and *max* can only be one. *Const*$_0$, *const*$_1$, ..., *const*$_n$ are additional constraints used to specify any interdependencies that might exist between different combinations of task selections. Notice the first form is merely a specialization of the second form. When the third form is used, the access information contained in the elaboration specifies the group of people with the appropriate access privileges that may perform the USER-SELECT operation.

4.4 Fact Base Management

While constraints play a major part in the management of the procedural nature of the development process, considerable information is represented outside the scope of the Constraint Propagator. This information is used to control user access to activities, artifacts, tools, and methods. The Fact

Base Manager is the component of the Framework Processor that manages this information.

The information found in the fact base comes from the framework definition and is extracted from the fact statements found in the Elaboration specifications of UOBs (see Section 3.2.1.2.3). During this extraction process, the facts are passed through the Validator and then stored in the fact base. Once there, the information is available to the Fact Base Manager to assist in responding to access queries.

These queries are really the heart of the functionality provided by the Fact Base Manager. In the development framework, the framework designer has specified what users should have access to certain tasks and artifacts. During interaction with the Framework Processors, users find themselves working in many different contexts within the development effort (i.e., they work on many different tasks during the same session). It is up to the Framework Processor to detect these context switches and to determine if the user is authorized to make the switch. The way the Framework Processor accomplishes this is by constructing a statement that represents the user's new context and querying the Fact Base Manager to see if that context statement is valid for that user. The response from the Fact Base Manager will determine whether the Framework Manager will allow the user to enter the new context.

As this discussion has shown, the Fact Base Manager, in support of the Framework Manager, stores and retrieves user authorization and access information. In doing so, the Fact Base Manager serves as a nice complement to the dynamic nature of the Constraint Propagator by providing support for managing and accessing static framework information.

4.5 Development Process Control

A conceptual view of the Framework Manager operation has already been presented. This section focuses on detailing the complex interactions that occur during the controlling of a system development process as highlighted in Figure 25.

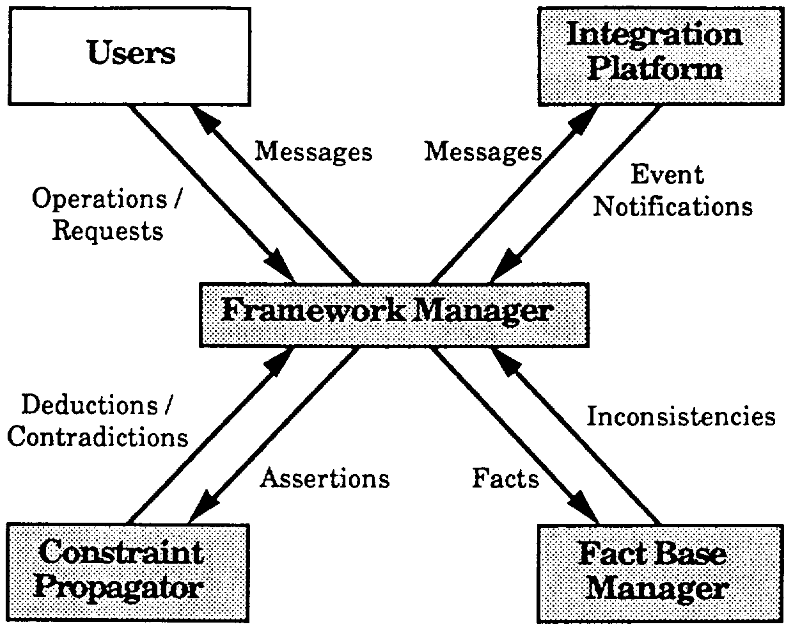


Figure 25. Development Process Control

For the sake of discussion, assume that an event notification has just been received by the Framework Manager from the Integration Platform via the Platform Interface Manager. The information contained in an event notification is illustrated in Table 2.

Project:	CAD Modeler
Task:	Requirements Definition
User:	John Smith
User role:	Analyst
Assertion:	(ARTIFACT "Requirements Document" RELEASED)

Table 2. Example of an Event Notification

Upon receipt of an event notification, the Framework Manager performs a two-tiered event validation process to determine the integrity of the event notification. The first validation step involves querying the fact base via the Fact Base Manager to determine for the given three tuple <project, task, user role> whether the specified user has the appropriate access privileges. If the access privileges are invalid, the event notification is logged as invalid and the appropriate framework administrators are notified. Otherwise, the second validation step is performed. This step involves matching the given event assertion against the valid assertions in the fact base. If an inconsistency exists the user is notified of the situation along with the framework administrators.

After passing event validation, the event assertion is passed to the Constraint Propagator by the Framework Manager. At this point, the Constraint Propagator makes deductions based on this new piece of information. For example, this new assertion could represent the final completion criteria for a task. Thus, the Constraint Propagator would deduce the completion of one task possibly asserting the activation of another task.

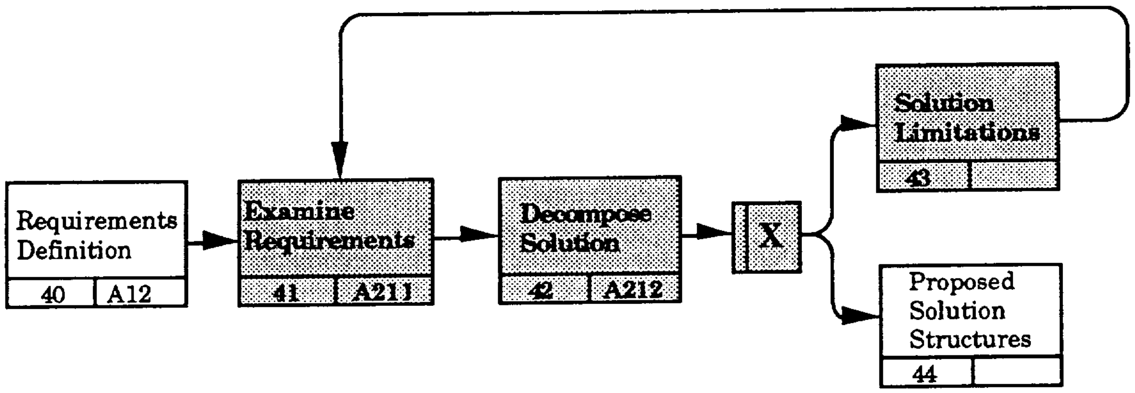


Figure 26. Loop Back Retractions

Two special case assertions must be handled by the Constraint Propagator and the Framework Manager, respectively. The first special case is caused by loop backs in an IDEF3 description and is handled by the Constraint Propagator. The loop back causes the Constraint Propagator to reassert the activation of a task that has already been activated. In fact, the task is still considered completed by the Constraint Propagator. To force the framework user to redo a task, the Constraint Propagator must retract any assertions made by the tasks occurring from the loop back point. For example, any assertion made during the "Examine Requirements" task, the "Decompose Solution" task, the "XOR" junction, and the "Solution Limitations" task would have to be retracted by the Constraint Propagator as highlighted in Figure 26.

A potential problem in retracting information from previous iterations to allow for new iterations is that the history of the development process represented by assertions made during the previous iterations is lost. Another possible approach to this problem is to reinstantiate (i.e., recasting the elements in the process description into new constraints) the development process. With this approach, though, it is difficult to determine how much of the development process to reinstantiate and to decide what assertions from previous instantiations carry over to the new instantiation. The difficulties in both approaches result directly from the complexities introduced by loop backs in IDEF3 descriptions. The retraction approach represents the initial attempt to be made at addressing these difficulties. However, the second approach was introduced to show a potential alternative in case the retraction based solution proves to be insufficient for the needs of the Framework Processor.

The other special case occurs when the Constraint Propagator asserts a USER-SELECT relation. Upon making a USER-SELECT assertion, the Constraint Propagator must inform the Framework Manager of the assertion. The Framework Manager uses the information specified in the USER-SELECT relation to query a user with the appropriate access privileges, as specified in the access privileges in the associated elaboration, for the selection of the branches to pursue following a fan-out junction. After acquiring the information, the Framework Processor makes a series of SELECT relation assertions based on the user supplied information.

4.6 Session Management

The Session Manager component of the Framework Processor is responsible for managing the interaction between users and the Framework Manager. Three major classes of users have been identified and appropriate interfaces and functions will have to be provided by the Session Manager to support these users. The following sections will describe the interactions and functions that will be supported by the Framework Processor concerning each of these user classes.

4.6.1 Project Team Members

Project Team Members represent the largest class of users of the Framework Processor. These users are the people that perform the tasks and duties required to produce the software systems. In essence, these are the people being managed by the Framework Processor. The following subsections will describe how these users will interact with the Framework Processor and how the Framework Processor will aid them in performing their tasks.

4.6.1.1 Logging Into the Framework Processor

When the user begins a session on the Framework Programmable Platform, the user must provide information to the Platform so that the system will know in which context the user intends to operate. As a result, for a user to attempt to login, the following information will have to be specified:

1) the user's system id;
2) the user's password;
3) the Project on which the user intends to work; and
4) the User Role under which the user intends to work.

Once this information has been provided, the Session Manager will begin the authorization process. This involves determining if this user, serving in the specified role, has the authority to work on the specified project. This authorization check is performed by comparing the information provided by the user against the user profile information (see Section 3.1.6) maintained by the framework. If the comparison succeeds, the user is granted access to the system; if not, the user is asked to change the information or exit the system.

This authorization procedure is really no different than the procedures on any other computer system. The additional information regarding the user role and project are required to reduce the possibility of confusion by the user. It is possible that one user might serve in several different roles for a specific project or might serve the same role, but for many different projects. By requiring the user to specify the selected information initially, the system is establishing the context in which the user will be operating. This characteristic should prevent the user from making unintentional mistakes during the development effort.

4.6.1.2 Project Visualization

When the authorization procedure is completed successfully, the Framework Processor will present the user with screens that will display the status of the project under which the user logged in. These displays are intended to provide visual clues regarding the status of the project and to guide the user to where current work activities should proceed.

Figure 27 presents a display of the Situation Classification Framework chart. Each of the cells are shaded to indicate a status or access authorization for that cell. Notice that certain cells are shaded black. This indicates that the situation represented by that cell never arises during the development process at this organization or is not used for this particular project. This fact is represented in the top level process description by the fact that those cells do not occur in the process description. On the other hand, the white cells represent those cells that the user, based on the role type, is authorized to participate in and those cells can be examined more closely by the user. The other cells are inaccessible to the user as they have already been completed, the user does not have the proper authorization, or both.

	Data	User	Function	Network
Objective/ Scope	▩	▨	▨	▨
Domain Model	▩	▒		▨
Model of the Business	▒	▒		
Model of the Information System			▨	■
Technology Model	▨			
Detailed Represen- tations	■			▨

▒ Completed ▩ Unauthorized, but Completed

▨ Unauthorized ■ Not Included in This Project

□ Authorized, but Not Completed

Figure 27. Interface for the Situation Classification Framework

While the interface for the Situation Classification Framework shows the cells where the user can participate, the interface does not indicate the sequence in which those cells should be examined. For this, the System Development Framework must be accessed. The display shown in Figure 28 shows how a diagram in the System Development Framework will appear to the user. Again, shading is used to provide visual cues as to the state of a particular activity. Those activities that are not shaded are the activities to which the user has complete access.

It should be noted that, while this interface prohibits users from examining certain cells or activities based on a user's role type, it will be possible for the user to browse the entire framework to provide the context for activities in which the user is involved. This browsing capability can be provided by defining a special project or role types that supports unlimited access authority. However, there will be no way for the user, while assuming one of these dummy roles, to actually perform any task associated with the project. The dummy project or role types exist only to serve as a browsing mechanism for the entire process development framework. In this respect, the dummy project or role types could serve as the means for training new team members in the development process of the organization.

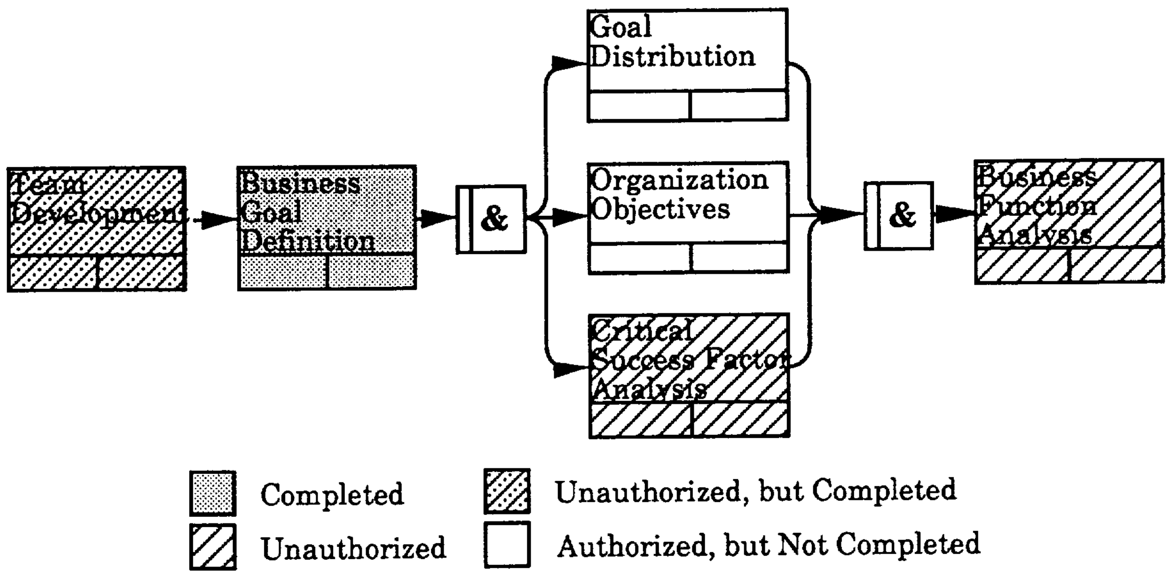

Team Development
Business Goal Definition
&
Goal Distribution
Organization Objectives
Critical Success Factor Analysis
&
Business Function Analysis
Completed
Unauthorized
Unauthorized, but Completed
Authorized, but Not Completed

Figure 28. Interface for the System Development Framework

The construction of these interface displays will require the Session Manager to interact closely with the Framework Manager. The reason for this is that these displays represent the current state of a development project. This is precisely the information that the Framework Manager monitors through the constraint and fact base. Essentially, what will happen will be a parse of the process description provided by the framework. For each cell or activity in the framework, a two step authorization / status check will be performed. For status purposes, the two steps are independent of each other and no specific processing order is required. One step will be a comparison against the fact base to determine if the user is authorized to access a particular cell or process activity. The user-id and user role of the user will be compared against the access privileges defined for that cell or task (see Section 3.2.1.2.3). This comparison will result in allowing or disallowing access to the contents of the cell or activity.

The other step in this process involves determining the state of the cell or activity in question (i.e., determining whether the activity has been completed or not). This is accomplished by querying the constraint propagator for the status of the the particular activity. Assume the system is attempting to determine the state of the "Business Goal Definition" activity shown in Figure 28. The Session Manager, through the

Framework Manager, would query the constraint base with the following form:

(QUERY (DONE "Business Goal Definition"))

This form, when evaluated by the Constraint Propagator, would determine whether the relation (DONE "Business Goal Definition") is considered to be true or false. If the result is true, then the activity is completed; otherwise, the activity is in progress or has not started, depending on the state of the activities preceding "Business Goal Definition."

With the results of these operations, the status displays can be generated and presented to the user. However, the construction of these status displays could require a considerable amount of processing time. For this reason, it may be beneficial to generate the status display for a process description only when the user attempts to access that description. Also, since the process descriptions can have many levels (through process decomposition), when an activity is found for which the user has no authority to view, there is no need to generate the status displays for the decomposition of that activity. A decision on exactly how these displays will be generated will be made after experimentation with the efficiency of the access authorization and status querying procedures.

4.6.2 Project Managers

The second class of users of the Framework Processor are the project managers. These users are unique in that they can fill two functions while interacting with the Framework Process. First, the project manager can be serving as a normal user as in the previous section, but filling the role of project manager. In this role, the project manager requires no new capabilities. However, it is in the second function of project instantiation and administration that additional capabilities are required. The instantiation process performs system initializations required to prepare the Framework Processor for the development of the project and the administration involves both monitoring and managing of the system resources used to represent the project instantiation. As a result, the project manager will have the ability to perform more powerful, yet potentially more dangerous, operations.

To begin with, the Session Manager must provide functionality by which the project manager can actually instantiate a new project. Besides providing the project manager with the ability to name the project, functionality for assigning team members to the project is required. Additionally, minor modifications to the framework can be established for this project. The extent of these modifications, at present, is to allow the project manager to select a subset of the allowable methods and tools defined as part of the framework. Once these modifications are completed, the project can be instantiated and development can begin.

At certain times after instantiation, it may become necessary to modify the instantiated project. For example, assume that a project manager established that Task A of a specific project had to be signed off by User John-Doe. However, since the instantiation of that project, but before the completion of Task A, User John-Doe resigns and leaves the company. The project manager must have the ability to modify the project information. The John-Doe example would require that the project manager be able to modify the completion criteria for Task A. This is easily accomplished as the Constraint Propagator supports the retraction of assertions. To correct the situation created by John-Doe's resignation, only the following statements need to be evaluated to remove John-Doe and assign the sign-off responsibility to Jane-Smith.

```
(RETRACT   (->   (USER-SIGN-OFF   John-Doe)
                 (DONE            Task-A)))

(ASSERT    (->   (USER-SIGN-OFF   Jane-Smith)
                 (DONE            Task-A)))
```

While the Constraint Propagator can perform these modifications, the Session Manager provides the user interface for making these modifications.

4.6.3 Framework Administrators

Perhaps the least common, but potentially most important, user of the Framework Processor is the framework administrator. This user is responsible for the installation of a specialized framework at a particular site. It is during this installation process that the framework will undergo the validation procedures described in Section 4.2. In this situation, the Session Manager interacts with the Validator, through the Framework Manager, to detect and correct any inconsistencies in the framework definition.

5 Status and Future Directions

While effort is being expended to make the Framework Processor an entity independent of any specific integration platform. However, the Framework Processor is evolving as part of the overall FPP. The purpose of this section is to describe how the Framework Processor relates to the other components of the FPP architecture, identify the subset of the FPP requirements addressed by the Framework Processor, and identify areas where future work concerning the Framework could be directed.

5.1 Framework Processor and the FPP

While the Framework Processor is being designed to be independent of any specific software development environment, the Framework Processor is being included in an integration platform being developed as part of the FPP effort. Figure 29 shows the current architecture of the Framework Programmable Platform. The diagram represents the functional architecture where each box in the diagram represents a functional unit and the links between the boxes represent communication between those functional units. It should be noted that this architecture is a derivative of the Design Knowledge Management System platform architecture that KBSI is currently developing for the Air Force [DKMS 90]. The approach in the FPP project has been to leverage off of the DKMS effort by first adopting the integration strategy of the DKMS and then layering the framework programmability on top. A discussion of this architecture is required to set the context for how the Framework Processor fits in with other components of the FPP. The remainder of this section will be dedicated to this discussion.

The *FPP Session* and *Application* represent the external interfaces to the FPP. The FPP Session serves as the users direct link to the FPP environment while the Application component represents the interface through which applications would access FPP functionality. These two components would provide user and application interfaces to the Integration Mechanism, the Data Object Manager, and the Facilitator.

The *Integration Mechanism* is responsible for monitoring and controlling the generation and execution of integration service plans. The idea behind the services approach is based on the view that computer tools and utilities provide services to users. The Integration Mechanism provides the means for defining the services provided by tools and support for the automatic execution of those services. See [FPP 91] for a more detailed discussion of the integration services concept.

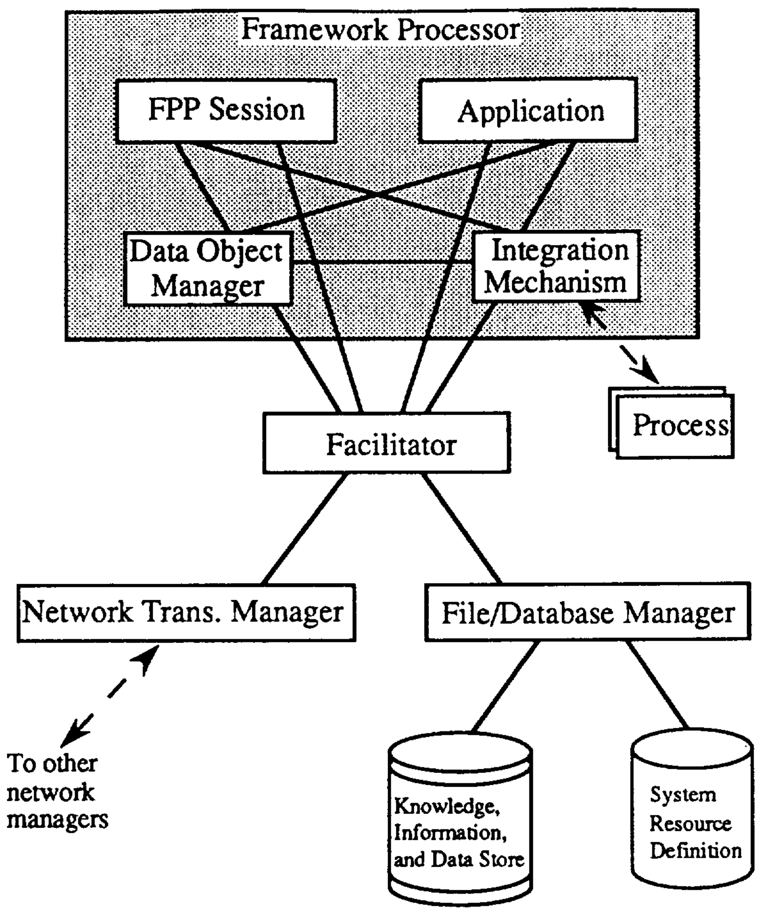


Figure 29. FPP Architecture

The *Data Object Manager* (DOM) is responsible for the management of life cycle data artifacts. In managing life cycle artifacts, the DOM will provide functionality for registering data artifacts in the repository and to maintain access control over the artifacts. The DOM will also include versioning and configuration management functionality and will be used to manage system resource artifacts. The need to do this stems from the fact that the FPP operates in a distributed environment. The namespace for the DOM, as a result, will be distributed across all nodes of the platform. To take advantage of this single repository and eliminate the need to store system resource data on every machine, the DOM will manage these artifacts.

The *Facilitator* serves as a dispatcher of messages between higher level components (DOM and Integration Mechanism) and the lower level components (Data Managers and Network Transaction Manager). This separation between higher and lower level components is required since the Data Managers and Network Transaction Managers will be more machine dependent than the more portable DOM and Integration Mechanism. The Facilitator will provide a common interface between these two levels so that the impact of changes in one level will be reduced, if not eliminated, in the other level.

The distributed nature of the FPP also makes the Facilitator necessary. When accessing data, whether that data resides on the local machine or on a remote machine should be transparent to the DOM. To hide the location of the data from the DOM requires an intermediate party to parse the data id and route the data query to either the appropriate machine (through the Network Transaction Manager) or the appropriate data manager on the local machine.

The Facilitator also serves as the interface between the FPP and *File/Database Managers* running on the machine. This architecture allows the host file system and different database managers to be used for FPP data storage. The location of the data will be encapsulated in the data ID. The Facilitator will extract the location and formulate a query based on the query structure of the database manager. The query will then be passed to the appropriate manager where the data will be collected and returned to the Facilitator.

The *Network Transaction Manager* is responsible for sending and receiving network operations for the local machine. The operations might include data queries or updates to database managers running on other machines, request for service execution on remote machines, or simple network file transfer operations. The gaol of the NTM is to provide a common networking interface between different FPP nodes that provides a higher level of abstraction than the many existing networking protocols.

The *Knowledge, Information and Data Stores* will store the data artifacts being maintained and controlled by the FPP. These stores will not only contain the data artifacts themselves but will also include data and knowledge necessary to manage those artifacts. The management information will include access control information, audit trail information, configuration and version control information, as well as dependency relationship information. Rules and constraints for the manipulation and management of these data artifacts will be established by the defined framework.

The *System Resources Definition* repository will contain information required by the FPP to operate. This would include knowledge about tools, applications, services, hosts, and database servers operating under the FPP as well as users of the FPP. Additionally, information extracted from the framework and used by the Framework Processor would be stored in these resource definitions.

The *Framework Processor* will mainly interact with the higher level components of the FPP (FPP Session, Application, DOM, and Integration Mechanism). Lower level functionality will be accessed by the Framework Processor through the normal interfaces defined as part of the FPP. As was described in Section 2.2, cooperation between the Framework Processor and the Integration Platform will be required to enforce the policies specified in the framework. So, instead of a Framework Processor controlling the operation of every component, each component will access the information about the framework to ensure that the constraints of the framework are not violated. In light of this role, the framework processor would simply respond to queries from the various components about the framework contents. In addition, the various components would send messages to the Framework Processor to indicate the occurrence of events during the evolution of active projects.

5.2 Requirements Matrix

With an understanding of the roles the various components of the FPP are expected to play, it is possible to establish the requirements of the FPP that have been addressed by the Framework Processor design. The matrices in Figures 30 and 31 relate the requirements [FPP 90b] satisfied by the design of the Framework Processor to the actual component of the Framework Processor that satisfies the requirement. In many cases, a requirement is satisfied through a combination of several Framework Processor components.

It should be noted that the occurrence of a requirement in the requirements matrix does not imply that the requirement is completely satisfied by the capabilities of the Framework Processor. As the previous section showed, the FPP is a complex system with many functional components. It is only through the development of all these components that the FPP requirements will be met. The purpose of the requirements matrix is to identify those areas of the FPP requirements that this component, the Framework Processor, will address.

5.3 Open Issues

While the Framework Processor addresses a considerable number of the requirements specified for the FPP, certain areas could still be addressed by the components of the Framework Processor. A brief discussion of these topics is presented below.

Versioning and Configuration Management

Currently, the DEFARTIFACT form of the Definition Component of the framework does not address versions and configurations. As the DOM is expected to support the capture and management of versions and configurations of artifacts, it would be useful to capture an organization's policies on versions and configuration management in the framework definition and to use that information in the development process control.

	Session Manager	FW Manager	Validator	Constraint Propagator	Fact Base Manager
2.1.1		√			
2.1.2		√			
2.1.3		√	√	√	√
2.1.4		√	√	√	√
2.1.5		√	√		
2.1.6	√				
2.1.7		√		√	
2.1.8				√	√
2.1.9	√	√			
2.1.10	√	√	√	√	√
2.2		√		√	√
2.5.1				√	√
2.5.2		√			√
2.6.1		√			
2.6.2		√			√
2.9.1		√			
2.9.2		√			
3.1.1	√				
3.3.1.2.2		√			
3.3.1.4.3		√		√	√
3.3.1.4.4		√		√	√
3.4.1.1		√			√
3.4.1.2.1	√	√		√	√
3.4.1.2.2	√	√		√	√
3.4.1.2.3	√	√		√	√
3.4.1.2.4	√	√		√	√
3.4.1.2.5	√	√		√	√
3.4.1.5.1					√
3.4.1.6.1		√			√
3.4.1.6.2		√		√	√
3.4.2.1	√	√			√

Figure 30. Framework Processor Requirements Matrix

	Session Manager	FW Manager	Validator	Constraint Propagator	Fact Base Manager
3.4.2.2	√	√	√	√	√
3.7.2.1		√		√	√
3.8.3	√	√			
4.1.1	√	√			√
4.1.5	√	√			√
4.3.1.1.1				√	√
4.3.1.1.2		√			
4.3.1.2.1		√	√		√
4.3.1.2.2		√	√	√	
4.3.2.5		√		√	√
4.6.1		√			√
4.7.4				√	√
4.7.6		√			√
4.7.7		√			√
4.11.1.1		√			√
4.11.1.2		√			√
4.11.1.3		√			√
5.1.5	√	√			√
5.1.6	√	√			√
5.2	√				√

Figure 31. Framework Processor Requirements Matrix (continued)

Artifact Types

Another issue that could be addressed by the Framework Processor is the definition of artifact types. The ability to do this would cut down on the amount of definitional information required by allowing information to be captured in the type definition and then to attach that information to artifacts as the artifacts are defined.

User Groups

With User Groups, the idea would be to arrange user roles defined in the framework into groups of related roles. It would then be possible to define access constraints on user groups instead of on every user role that is a part of that group. Again, this capability would simplify the process of framework definition.

Constraint Specification Language

The representation of constraints in the framework definition is currently specified in the internal format to be used by the Constraint Propagator component of the Framework Processor. The issue of a Constraint Specification Language would involve defining a language that would be more accessible to the framework designer and defining the rules for translating those constraints into their internal format. An initial candidate for this purpose is the Information Systems Constraint Language (ISyCL).

Internal IDEF3 Representation

As was mentioned in Section 3.2.1, a textual representation for IDEF3 process descriptions was not defined as part of the Framework Processor design. The point of defining this representation would be to allow the Framework Processor to interface with many different IDEF3 tools. It is expected that an initial representation for IDEF3 will result from the IDEF3 ISyCL Metalanguage task being performed as part of the Information Integration for Concurrent Engineering project at KBSI and this representation will be used by the Framework Processor.

5.4 Future Directions

As the initial designs of the Integration Mechanism [FPP 91] and Framework Processor have been completed, the next portion of the FPP project will focus on the design of the remaining components of the FPP Architecture (see Section 5.1). A particularly important component of that architecture will be the Data Object Manager. It is expected that the operation of that component will have the most impact on the Framework Processor as well as the Integration Mechanism.

During the design of these additional components, the Framework Processor design will be reevaluated and examined for required additions and modifications. This examination will allow us to attempt to address the issues presented in the previous section and to also take advantaged of the functionality defined as part of the design of the remaining FPP components.

6 Bibliography

[Aho 86] Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[DKMS 90] *A Design Knowledge Management System (DKMS)*, SBIR Phase I Final Report, April 1990, Knowledge Based Systems, Incorporated. Contract F41622-89-C-1018, AFHRL, WPAFB.

[FPP 90a] *Framework Programmable Platform for the Advanced Software Development Workstation: Concept of Operations Document.* Report to NASA and University of Houston-Clear Lake by Knowledge Based Systems, Inc under subcontract SE.37, NCC9-16. September, 1990.

[FPP 90b] *Framework Programmable Platform for the Advanced Software Development Workstation: Requirements Document.* Report to NASA and University of Houston-Clear Lake by Knowledge Based Systems, Inc. under subcontract SE.37, NCC9-16. November, 1990.

[FPP 91] *Framework Programmable Platform for the Advanced Software Development Workstation: Integration Mechanism Design Document.* Report to NASA and University of Houston-Clear Lake by Knowledge Based Systems, Inc. under subcontract 077, NCC9-16. June, 1991.

[Mayer 90] Mayer, R., Menzel, C., and Mayer, P., *IDEF3 Technical Report*, Knowledge Based Systems Laboratory, Texas A&M University, March 8, 1990.

[Menzel 91] Menzel, C., Mayer, R., and Edwards, D., *IDEF3 Formalization Report*, Knowledge Based Systems Laboratory, Texas A&M University, March 6, 1991.

[Zachman 86] Zachman J.A., *A Framework for Information Systems Architecture*, IBM Los Angeles Scientific Center, G320-2785, March 1986.

PRECEDING PAGE BLANK NOT FILMED

PAGE 72 INTENTIONALLY BLANK

A Appendix A - Acronyms

ASDW	Advanced Software Development Workstation
DOM	Data Object Manager
FP	Framework Processor
FPP	Framework Programmable Platform
NTM	Network Transaction Manager
SCF	Situation Classification Framework
SDF	System Development Framework

PRECEDING PAGE BLANK NOT FILMED

PAGE 74 INTENTIONALLY BLANK

B Appendix B - Framework Definition Grammar Specification

This appendix contains the complete lexical and grammar specification for the forms that are part of the Definition Component and the Elaboration Specification.

Lexical Conventions

This section describes the lexical conventions used in the definition of the specifications. Where necessary a regular definition [Aho 86] has been provided to explicitly and unambiguously express a lexical item. The lexical conventions are:

1) A semicolon (';') starts a comment and the comment is terminated by the end of the line.

2) Spaces (' ') between tokens are optional. However, keywords must be surrounded by spaces and newlines.

3) An *identifier* is made up of a letter followed by letters, digits, or underscores. The regular definition form of an *identifier* is as follows:
   letter ::= [a-zA-Z]
   digit ::= [0-9]
   identifier ::= *letter* (*letter* | *digit* | _)*

4) An integer is composed of optionally a plus or minus sign followed by at least one digit. The integer regular definition is as follows:
   digits ::= *digit digit**
   integer ::= (+ | - | e) *digits*

5) A real number may be represented either in decimal notation or scientific notation. Therefore, a real number is represented by the following regular definition:
   fraction ::= . *digits* | e
   optional-exponent ::= ((E | e) (+ | - | e) *digits*) | e
   real ::= (+ | - | e) *digits fraction optional-exponent*

6) A string is delimited by double quotes ("") containing any printable ASCII character.

PRECEDING PAGE BLANK NOT FILMED

PAGE 76 INTENTIONALLY BLANK

Grammar Conventions

Shown below are the conventions for the grammar of the specifications. The grammar is specified by listing its productions, with the production for the start symbol listed first.

1) *non-terminal* - Non-terminals symbols are represented in italics.

2) **terminal** - Terminal symbols are represented in bold. They represent keywords in the language. The parenthesis contained in the grammar are part of the specification. They are considered to be terminal symbols. However they will not be in bold.

3) An expression is made up of terminals, non-terminals, and other complex expression built from rules 4 through 7.

4) { *expression* | *expression* | *expression* } - The vertical bar ('|') represents a selection of one and only one item from the set of alternatives.

5) { *expression* }? - A question mark ('?') indicates that the expression can occur zero or one times.

6) { *expression* }+ - A plus sign ('+') indicates that the expression can occur one or more times.

7) { *expression* }* - An asterisk ('*') indicates that the expression can occur zero or more times.

framework-definition ::= *definitional-forms*
elaboration-forms

definitional-forms ::= {*declarations*}*

declarations ::= *defartifact* |
defuserole |
defproject |
defuser |
deftool |
defmethod |
defrelation

defartifact ::= (**defartifact** **:name** *identifier*
:tools *tool-list*
:states *states-list*
:method *method-list*)

defuserole ::= (**defuserole** **:name** *identifier*)

```
defproject ::=      (defproject  :name identifier
                                 :access access-list)

defuser ::=  (defuser      :name identifier
                           :password string
                           :projects project-list)

deftool ::=  (deftool      :name identifier
                           :version string
                           :formats format-list
                           :methods method-list)

defmethod ::=        (defmethod  :name identifier )

defrelation ::=      (defrelation  :name identifier )

format-list ::=      ( { format-name }+ )

format-name ::=      identifier

project-list ::=     ( { tool-name }+ )

project-name ::=     identifier

elaboration-form ::=       { elaboration }*


elaboration ::=      facts
                     constraints

facts ::=            { acc-role }+
                     { acc-art }*

acc-role ::=         (access-role :all :all) |
                     (access-role role-type :all) |
                     (access-role role-type :except access-list) |
                     (access-role role-type :only access-list)

acc-art ::=          (access-artifact artifact-name
                                  :tool tool-list
                                  :method method-list
                                  :state  state-name)

access-list ::=      ( { user-name | role-type }+ )

role-type ::= identifier

user-name ::=        identifier
```

artifact-name::= *identifier*

tool-list ::= ({ *tool-name* }+)

tool-name ::= *identifier*

method-list ::= ({ *method-name* }+)

method-name ::= *identifier*

state-name ::= *identifier*

constraints ::= { *completion-criteria* | ε }
{ *selection-criteria* }*
{ *general-constraints* }*

completion-criteria ::= *constraint-expression*

constraint-expression ::= (**and** { *constraint-expression* }+) |
(**or** { *constraint-expression* }+) |
(**xor** { *constraint-expression* }+) |
(**not** { *constraint-expression* }+) |
relation

selection-criteria ::= (**->** *constraint-expression selection-expression*)

selection-expression ::= (**select** *process-name*) |
(**and** (**select** *process-name*)
{ (**select** *process-name*) }+) |
(**user-select** *number number*
{ *constraint-expression* }*)

process-name ::= *identifier*

general-constraints ::= (**->** *constraint-expression constraint-expression*)

relation ::= (*relation-name* { *val* }*)

val ::= *identifier*

relation-name ::= *identifier*